UNIVERSIDADE NOVA DE LISBOA

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Electrotécnica

Comunicações intra- e inter-circuito de componentes especificados com Redes de Petri

Por:

Ricardo Wolffensperger Ferreira

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Professor Doutor Luís Gomes

Lisboa

2010

Agradecimentos

Este trabalho é a finalização de uma etapa importante da minha vida. Por esse motivo quero agradecer a todos os que contribuíram para ter chegado a esta fase e sem os quais esta tarefa teria sido muito mais difícil.

Ao Prof. Doutor Luís Gomes, por toda a disponibilidade e apoio dado. Mesmo quando as coisas me corriam menos bem, existiu sempre um incentivo e uma orientação do caminho certo a seguir. Não só tornou possível esta dissertação, como contribuiu para a minha formação.

À Prof. Anikó Costa, que me ajudou a modelar alguns exemplos de aplicação e a perceber melhor o funcionamento da ferramenta SPLIT.

Ao Prof. Doutor Ricardo Gonçalves, por me ter feito o convite e confiado no meu trabalho para ser monitor da disciplina de Sistemas Lógicos I.

Ao Prof. Doutor Celson Lima, Nuno Dias e Gonçalo de Castro, por, na nossa representação, em nome da faculdade e do país, na participação do concurso *imaginecup* da Microsoft, me terem ajudado a melhorar a minha metodologia de trabalho e ensinado o que é um verdadeiro trabalho de equipa.

Ao Prof. Tiago Cardoso, com quem me iniciei nas actividades extracurriculares, onde aprendi bastante e me deixou a força necessária não só para concorrer novamente ao *imaginecup,* como para os desafios que se advinham. E desde então tem contribuído na minha formação.

A todos os amigos, e colegas, com quem tenho trabalhado ao longo destes últimos tempos.

À minha família e amigos que toleraram a minha ausência, em particular à minha mãe que suportou o intolerável para que este dia existisse, e ao meu pai.

À Nanoca, por todo o seu apoio, sua compreensão e encorajamento constantes.

Sumário

Este trabalho tem como objectivo a apresentação de uma solução que possa vir a ser obtida através da geração automática de código para a interligação de componentes resultantes da partição de um modelo expresso em Redes de Petri – RdP – em diferentes plataformas.

A interligação proposta recorre a uma solução *Network-on-Chip* – NoC – com suporte a comunicações intra- e inter-circuito, baseada no protocolo RS-232 (embora possa funcionar a ritmos de transmissão mais elevados). O resultado obtido foi validado através da implementação em plataformas reconfiguráveis da Xilinx, FPGA Spartan3 e Virtex-II, e em microcontroladores de baixo custo, PIC 18F4620 da Microchip.

Considerando uma topologia em anel e utilizando o protocolo série RS-232 (não só suportado pela generalidade dos microcontroladores como por qualquer dispositivo externo que o suporte) serão introduzidas as regras e considerações necessárias para que seja possível gerar este tipo de soluções.

Finalmente, serão mostrados dois exemplos onde, através de um modelo inicial expresso em redes de Petri, posteriormente decomposto em vários submodelos, se apresentará o fluxo de desenvolvimento e o modo de aplicação das regras.

Abstract

This work presents a solution that might be obtained by automatic generation of code for the interconnection of components, obtained as a result of partition, of a model expressed in Petri Nets on different platforms.

The interconnection proposed solution uses a Network-on-Chip - NoC - supporting communications intra- and inter-circuit based on the RS-232 Protocol (although it can operate at higher transmission rates). The result was validated through implementation in the Xilinx reconfigurable platforms, Spartan3 FPGA and Virtex-II, and in low cost microcontrollers, Microchip PIC 18F4620.

Considering a ring topology and the protocol using the RS-232 (not only supported by most microcontrollers but also by any external device that supports them), rules and considerations needed to be able to generate this type of solutions will be presented.

Finally, two examples are shown where, through an initial model expressed as Petri nets further decomposed into several sub-models, will present the flow of development and how to apply the rules set.

Simbologia e notações

CDMA – *Code-Division Multiple Access*

CLB – *Configurable Logic Blocks*

CLK – *Clock*

CNN - *Central Coordination Node*

CRC – Cyclic redundancy check

FIFO – *Fisrt In – First Out*

FPGA – *field-programmable gate array*

GALS – Globally Asynchronous and Locally Synchronous

I^2C – *Inter-Intergrated Circuit*

ICE – Interface, Controlo e Entidade

ICS – *interface of controlled system*

IOPT – *Input-Output Place-Trasition*

IP – *Intelectual Property*

LED – *Light Emiting Diode*

MBD – *Model-based Development*

MISO – *Master Input, Slave Output*

MOSI – *Master Output, Slave Input*

NoC – *Network-on-Chip*

OSI – *Open Systems Initiative*

PCI – *Peripheral Component Interconnect*

PIC – *Programmable Interface Controller*

PNML – *Petri Nets Markup Language*

RdP – Redes de Petri

SATA – *Serial Advanced Technology Attachment*

SCL – *Serial clock*

SCLK – *Serial Clock*

SDA – *Serial Data*

SGML – *Standard Generalized Markup Language*

SoC – *System-on-Chip*

SoPC – *System-on-a-Programmable-Chip*

SPI – *Serial Peripheral Interface*

SPIN – *Scalable Programmable Integrated Network*

SS – *Slave Select*

TDMA – *Time-division Multiple Access*

UART – *universal asynchronous receiver/transmitter*

USB – *Universal Serial Bus*

VHDL – *VHSIC (Very High Speed Integrated Circuits) Hardware Description Language*

XGFT – *Generalized Extended Fat Tree*

XML – *eXtensible Markup Language*

XST – *Xilinx Synthesis Technology*

XUP – *Xilinx University Program*

Índice de Matérias

Índice de Figuras

Índice de Tabelas

1 Introdução

1.1 Enquadramento

A evolução no hardware tem vindo a possibilitar a integração de múltiplos componentes num único chip, tais como processadores, controladores dedicados e memórias, resultando na integração de um sistema completo num mesmo integrado. Este tipo de solução poderá recorrer a um circuito dedicado ou a um dispositivo reconfigurável. Estes sistemas são, normalmente, denominados *Systems-on-Chip* (SoCs) ou *Systems-on-a-Programmable-Chip* (SoPCs). É frequente o abuso de linguagem referindo como SoC as soluções SoPC.

Este aumento da complexidade dos sistemas faz com que as exigências de modelação sejam maiores, e que seja comum encarar a divisão do sistema em vários subsistemas inter-actuantes. No que se refere à comunicação entre estes subsistemas, o uso de ligações dedicadas é normalmente inviável, pois, apesar de oferecer melhor desempenho, ocupam demasiada área do ponto de vista da sua implementação SoC/SoPC, ou então necessitariam de interligações específicas quando se considerassem subsistemas implementados de forma heterogénea. Justifica-se, desta forma, a utilização de soluções de interligação dos subsistemas, através de uma rede dedicada, dando origem às normalmente designadas NoCs (*Networks-on-Chip*) apresentadas no capítulo seguinte.

Por outro lado, as metodologias actuais utilizadas no desenvolvimento de sistemas electrónicos, integrando componentes hardware e software, têm vindo a dar relevância crescente aos modelos (tendência para metodologias baseadas em modelos, MBD – *Model-based Development*). Ter um modelo que descreva adequadamente o sistema é uma mais-valia para o seu desenvolvimento e para a sua documentação. Se, a partir deste modelo for automaticamente gerado um código, poupar-se-á tempo gasto no desenvolvimento e, eventualmente, evitar-se-ão erros de implementação manual. A adopção de abordagens de desenvolvimento baseadas em modelos pode suportar melhorias claras no fluxo de desenvolvimento de sistemas, como referido em [7] e [8].

O projecto FORDESIGN [4], concluído recentemente, enquadra-se nestes objectivos, recorrendo à utilização de modelos expressos em RdP para a especificação do comportamento do sistema. Neste projecto foi desenvolvida uma classe de Redes de Petri denominada IOPT – *Input-Output Place-Transition* [9],

permitindo integrar dependências, no que se refere a sinais e eventos de entrada e de saída, suportando a modelação adequada de controladores.

1.2 Objectivos

O objectivo principal deste trabalho é o de apresentar uma solução desenvolvida para a interligação dos componentes gerados, recorrendo a soluções do tipo *Network-on-Chip* que possa, também ela, vir a ser obtida através de um gerador.

A solução de interligação será codificada em VHDL nas plataformas de implementação previstas para a validação da solução (onde se incluem as FPGAs da Xilinx das famílias Spartan-3 e Virtex-II), bem como em C para os sistemas externos com capacidade para suportar a solução de interligação pretendida (nomeadamente computadores de utilização geral e microcontroladores de baixo custo, sendo de referir os PIC da Microchip).

Desta forma, pretende-se obter uma solução (que possa vir a ser automaticamente gerada) que, embora com um desempenho limitado, possua uma elevada flexibilidade, baseada numa rede de comunicação série, e que permita gerar o código (VHDL e / ou C) da solução de interligação, partindo dos componentes associados expressos através dos seus modelos RdP-IOPT.

1.3 Estrutura

Este documento encontra-se dividido em várias secções: o presente capítulo, ao qual se segue o segundo capítulo, onde são descritos alguns conceitos e tecnologias que fundamentam o trabalho, os quais se revelam necessários para a sugestão de uma solução; no terceiro capítulo procede-se a uma descrição geral do sistema proposto; e o quarto capítulo apresenta dois exemplos de aplicação, onde são expostos e comparados cinco cenários para a interligação dos componentes, para que no quinto capítulo se conclua e se anunciem os trabalhos futuros.

2 Fundamentos e tecnologias

Tendo em conta o objectivo de interligar componentes gerados a partir de modelos, recorrendo a soluções do tipo *Network-on-Chip*, existiu a necessidade de: (1) conhecer melhor a classe de RdP desenvolvida no projecto FORDESIGN; (2) compreender o projecto FORDESIGN e quais as suas ferramentas; (3) explorar o que já existe sobre o tema das NoCs*;* (4) estudar os métodos existentes de interligar plataformas heterogéneas através de uma interface série utilizadas em sistemas de controlo e automação; (5) saber como interligar componentes, no caso de serem assíncronos entre eles; (6) entender o funcionamento de um *buffer* FIFO, necessário para a interligação dos componentes; e (7) estudar as ferramentas já existentes que cumpram os objectivos deste trabalho.

2.1 Redes de Petri

Como já referido, o objectivo deste trabalho é definir regras para gerar, de forma automática, o código de uma solução de interligação de vários módulos, anteriormente representados por modelos RdP-IOPT. Modelos esses que estarão expressos em PNML (*Petri Nets Markup Language*) [10].

2.1.1 Redes de Petri Input-Output Place-Transition

Nesta secção é apresentada a definição de redes IOPT, retiradas de [9], como uma extensão às classes de redes de Petri lugar-transição [11].

As características adicionadas ao que já tinha sido proposto em [12] foram: (1) definição de prioridades nas transições; (2) os lugares com um atributo de limite de marcas; (3) eventos de entrada e de saída; (4) dois tipos de sinais de entrada e de saída; (5) uma especificação explícita para um conjunto de transições com conflitos; (6) uma especificação explícita para um conjunto de transições síncronas; e (7) arcos de teste. Reproduzem-se seguidamente as seis definições principais relativamente à semântica de execução das redes IOPT, adaptadas de [12].

> *Definição 1 (Interface do sistema):* sinais e eventos, de entrada e saída, servem de interface entre o sistema e o controlador. Numa rede IOPT a interface do sistema controlado *(interface of controlled system - ICS*) é
>
> $$\mathrm{ICS} = (\mathrm{IS}, \mathrm{IE}, \mathrm{OS}, \mathrm{OE})$$

onde IS é um conjunto finito de sinais de entrada; IE é um conjunto finito de eventos de entrada; OS é um conjunto finito de sinais de saída; e OE é um conjunto finito de eventos de saída. Tem também que ser garantido $\mathrm{IS} \cap \mathrm{IE} \cap \mathrm{OS} \cap \mathrm{OE} = \emptyset$.

Definição 2 (Estado de entrada do sistema): considerando uma interface com as características da definição 1, o estado de entrada do sistema (*system input state - SIS*) é dado por:

$$\mathrm{SIS} = (\mathrm{ISB}, \mathrm{IEB})$$

sendo ISB um conjunto finito de estados de sinais de entrada, $\mathrm{ISB} \subseteq \mathrm{IS} \times \mathbb{N}_0$; e IEB um conjunto finito de estados de eventos de entrada, $\mathrm{IEB} \subseteq \mathrm{IE} \times \mathbb{B}$.

Definição 3 (redes IOPT): considerando uma interface com as características da definição 2, a rede IOPT é um conjunto

$$\mathrm{N} = (\mathrm{P}, \mathrm{T}, \mathrm{A}, \mathrm{TA}, \mathrm{M}, \mathrm{weight}, \mathrm{weightTest}, \mathrm{priority}, \mathrm{isg}, \mathrm{ie}, \mathrm{oe}, \mathrm{osc})$$

onde P é um conjunto finito de lugares; T é um conjunto finito de transições, disjunto de P; A é um conjunto de arcos, tal que $\mathrm{A} \subseteq ((\mathrm{P} \times \mathrm{T}) \cup (\mathrm{T} \times \mathrm{P}))$; TA é um conjunto de arcos de teste, tal que $\mathrm{TA} \subseteq (\mathrm{P} \times \mathrm{T})$; M é a função de marcação, $\mathrm{M}: \mathrm{P} \to \mathbb{N}_0$; Peso dos arcos, $\mathrm{weight}: \mathrm{A} \to \mathbb{N}_0$; Peso dos arcos de teste, $\mathrm{weightTest}: \mathrm{TA} \to \mathbb{N}_0$; A prioridade é uma função parcial que aplica transições a inteiros não negativos, $\mathrm{priority}: \mathrm{T} \to \mathbb{N}_0$; *isg* é uma função parcial de guarda de sinal de entrada que aplica transições a expressões booleanas, onde todas as variáveis são sinais de entrada, $\mathrm{isg}: \mathrm{T} \to \mathrm{BE}, \forall \mathrm{eb} \in \mathrm{isg}(\mathrm{T}), \mathrm{Var}(\mathrm{eb}) \subseteq \mathrm{IS}$(onde BE é o conjunto de expressões booleanas, Var(E) retorna o conjunto de variáveis numa dada expressão E); *ie* é uma função parcial de eventos de entrada que aplica transições a eventos de entrada, $\mathrm{ie}: \mathrm{T} \to \mathrm{IE}$; *oe* é uma função parcial de eventos de saída que aplica transições a eventos de saída, $\mathrm{oe}: \mathrm{T} \to \mathrm{OE}$; e *osc* é uma função para um sinal de saída de lugares para conjuntos de regras, $\mathrm{osc}: \mathrm{P} \to \mathrm{P}(\mathrm{RULES})$ ($\mathrm{RULES} \subseteq (\mathrm{BES} \times \mathrm{OS} \times \mathbb{N}_0), \mathrm{BES} \subseteq \mathrm{BE}$ e $\forall \mathrm{e} \in \mathrm{BES}, \mathrm{Var}(\mathrm{e}) \subseteq \mathrm{ML}$, ML é o conjunto de identificadores para cada marcação de lugar. Após um estado de execução, cada marcação de lugar tem associado um identificador que é usado quando o código gerado é executado).

Definição 4 (condição de activação): tendo uma rede N como a descrita na definição 3, uma interface de sistema como a especificada na definição 1 entre a rede e um estado de entrada do sistema conforme a definição 2, uma

transição t, sem conflitos estruturais, está habilitada a disparar num determinado "tick", se, e só se, as seguintes condições forem satisfeitas: (1) $\forall \mathrm{p} \in {}^{\cdot}\mathrm{t}, \mathrm{M}(\mathrm{p}) \geq \mathrm{weight}(\mathrm{p},\mathrm{t}), s$, com o conjunto de lugares de entrada interligados por arcos comuns; (2) $\forall \mathrm{p} \in {}^{\diamond}\mathrm{t}, \mathrm{M}(\mathrm{p}) \geq \mathrm{weightTest}(\mathrm{p},\mathrm{t})$, com o conjunto de lugares de entrada interligados por arcos de teste; (3) a *input signal guard* da transição *t* é avaliada como verdadeira para o sinal de entrada dado segundo, isg(t)<ISB>; e (4) $(\mathrm{ie}(\mathrm{t}), \mathrm{true}) \in \mathrm{IEB}$. Em contrapartida, quando a transição se encontra num conflito estrutural com outras transições, só está habilitada se tiver a prioridade mais elevada relativamente ao respectivo conjunto de transições habilitadas $\mathrm{CS}: \forall \mathrm{t}' \in \mathrm{CS}, \mathrm{t}' \neq \mathrm{t} \Longrightarrow \mathrm{priority}(\mathrm{t}') \leq \mathrm{priority}(\mathrm{t})$.

Definição 5 (Passo de rede IOPT): considerando uma rede N idêntica à da definição 3, uma ICS como assente na definição 1 e um SIS da definição 2. Considerando ainda que $\mathrm{ET} \subseteq \mathrm{T}$é o conjunto de todas as transições habilitadas como definido em 4. Então, Y é um passo em N se, e só se, estiverem satisfeitas as seguintes condições: $\mathrm{Y} \subseteq \mathrm{ET} \wedge \forall \mathrm{t}_1 \in (\mathrm{ET} \backslash \mathrm{Y}), \exists \mathrm{SY} \in \mathrm{Y}, ({}^{\cdot}\mathrm{t}_1 \cap {}^{\cdot}\mathrm{SY}) \neq \emptyset \wedge \exists \mathrm{p} \in ({}^{\cdot}\mathrm{t}_1 \cap {}^{\cdot}\mathrm{SY}), (\mathrm{weight}(\mathrm{p},\mathrm{t}_1) + \sum_{\mathrm{t} \in \mathrm{SY}} \mathrm{weight}(\mathrm{p},\mathrm{t}) > M(p))$.

Definição 6 (Ocorrência de passo e sucessor de marcação): tendo uma rede N com as características da definição 3, uma interface de sistema ICS como na definição 1 e um SIS como na definição 2, a ocorrência do *passo* Y na rede N retorna a rede $\mathrm{N}' = (\mathrm{P}, \mathrm{T}, \mathrm{A}, \mathrm{TA}, \mathrm{M}', \mathrm{weight}, \mathrm{weightTest}, \mathrm{priority}, \mathrm{isg}, \mathrm{ie}, \mathrm{oe}, \mathrm{osc})$ igual à rede N, com excepção do sucessor de marcação M' que é dado pela seguinte expressão:

$$\mathrm{M}' = \left\{ \left(\mathrm{p}, \mathrm{m} - \sum_{\mathrm{t} \in \mathrm{Y} \wedge (\mathrm{p},\mathrm{t}) \in \mathrm{A}} \mathrm{weight}(\mathrm{p},\mathrm{t}) + \sum_{\mathrm{t} \in \mathrm{Y} \wedge (\mathrm{p},\mathrm{t}) \in \mathrm{A}} \mathrm{weight}(\mathrm{t},\mathrm{p}) \right) \in (\mathrm{p} \times \mathbb{N}_0) \middle| (\mathrm{p},\mathrm{m}) \in \mathrm{M} \right\}$$

2.1.2 PNML – Petri Net Markup Language

Com base na descrição de [10], o PNML foi definido como um formato de representação de RdPs permitindo o intercâmbio entre ferramentas, independentemente da ferramenta e aplicação. É baseado em XML (*eXtensible Markup Language*), tal como outros formatos desenvolvidos para representar RdP, e existem princípios de flexibilidade e compatibilidade como a necessidade de não ser ambíguo.

A flexibilidade significa que o PNML deve poder representar qualquer tipo de RdP com as suas características e extensões específicas.

A ambiguidade é retirada do formato, assegurando que a RdP original e o seu tipo particular possam ser determinados unicamente através da sua representação PNML. Com este intuito, o PNML suporta a definição de diferentes tipos.

A compatibilidade significa que toda a informação possível deve poder ser trocada entre diferentes tipos de RdP. A fim de alcançar esta compatibilidade, o PNML tem vindo a desenvolver acordos para definir uma *label* especifica com um determinado significado.

2.1.3 Operação Net splitting

No trabalho publicado em [13] é apresentada uma operação em redes de Petri denominada *net splitting*, permitindo a partição de modelos de RdP, que tem como perspectiva a área de co-design hardware-software de sistemas embutidos.

Nessa proposta, a separação da rede inicial é realizada através de um conjunto de lugares, e transições dependendo da comunicação do sistema ou das infra-estruturas de comunicação. A Figura 2.1 mostra um exemplo de modelos a serem executados de forma paralela obtidos com a utilização da operação de *net splitting* e relativos a um sistema com emissor e receptor.

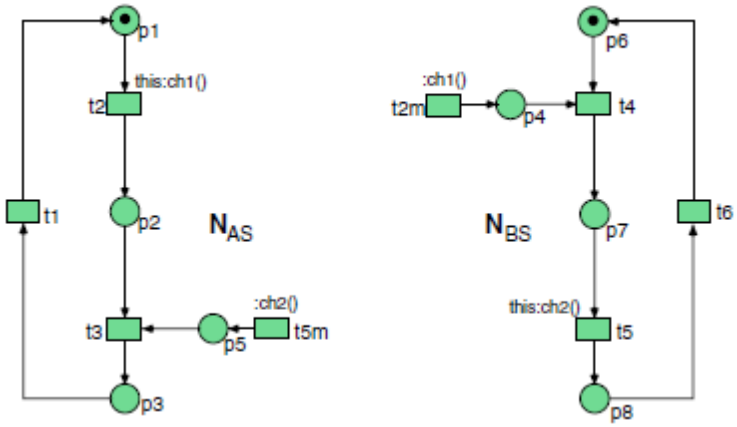


Figura 2.1 - Emissor - Receptor ligados através de canais síncronos [13]

Com o objectivo de se implementar o emissor e o receptor em plataformas distintas, ter-se-á que separar o modelo e encontrar / construir os respectivos (sub)modelos. O componente receptor tem que saber que o emissor enviou a mensagem, assim como o emissor tem que saber que o receptor recebeu a mensagem. Isto significa que cada modelo de um componente inclui os nós que

representam essa informação. Neste sentido, o receptor tem um lugar (p4 da Figura 2.1) representando que a mensagem foi enviada, e uma cópia da transição (t2 da Figura 2.1) para representar o envio da mensagem. A transição e a sua cópia (ou espelho) serão ligadas através de um canal síncrono. Da mesma forma o emissor inclui um lugar (p5 da Figura 2.1) e uma cópia da transição (t5 da Figura 2.1).

Em suma, a divisão do modelo inicial em componentes resultando o modelo da Figura 2.1 é realizado baseando-se no uso de transições com canais síncronos. O disparo da transição (por exemplo t2 da Figura 2.1), e sua cópia (por exemplo t2m da Figura 2.1), será sincronizado, a divisão das transições pode ser vista como uma conveniência gráfica, de forma a poderem ser identificados os componentes. Do ponto de vista de execução é completamente equivalente ao modelo inicial desde que o paradigma de execução síncrona entre os vários sub-modelos seja válido.

2.2 Projecto FORDESIGN

Distintos modelos de computação já provaram que se adequam à concepção de sistemas embutidos. Nomeadamente, os diagramas de estado e as redes de Petri. O projecto FORDESIGN, explorando as semelhanças entre formalismos, tenta, por um lado, reuni-los numa única metodologia e ferramentas de desenvolvimento relacionadas e, por outro lado, propõe o uso de uma classe de baixo nível de RdP [4] (RdP IOPT já referidas). Essa classe é então usada como linguagem de

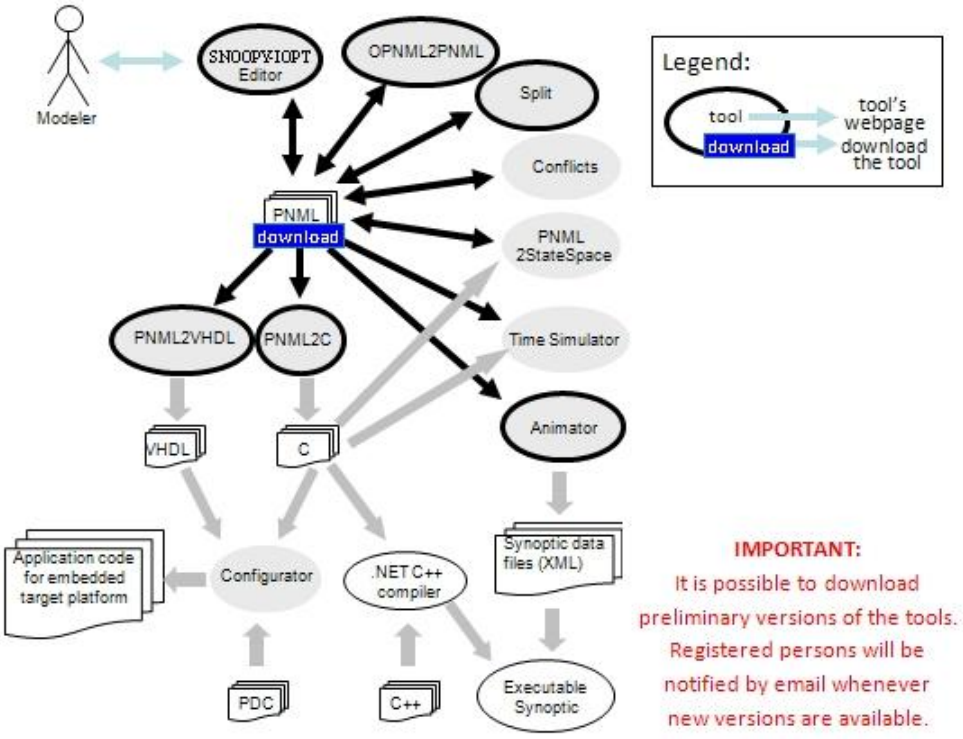


Figura 2.2 - Ferramentas em desenvovimento [4]

especificação, para a qual todos os formalismos referidos são traduzidos antes da sua verificação e implementação [4].

Definiu-se um conjunto de procedimentos, possibilitando o particionamento do modelo e identificação dos submodelos, reconhecido como componente. Cada componente pode ser mapeado em hardware ou software, de acordo com o seu custo específico, utilizando técnicas de co-design hardware-software.

A Figura 2.2, retirada de [4], mostra o conjunto de ferramentas que foram desenvolvidas, as quais se baseiam na representação de RdP através do padrão PNML (também já referido na secção anterior). As elipses contornadas a escuro são as ferramentas de que já existem versões preliminares.

No conjunto de ferramentas desenvolvido incluem-se:

- *Snoopy IOPT* – implementa um editor gráfico de RdP para a classe IOPT, apoiando especificações hierárquicas e modulares. Também suporta a representação PNML;
- *SPLIT* – é uma ferramenta de partição de um modelo IOPT em vários submodelos, usando canais de comunicação síncronos, posteriormente considerados como componentes em execução paralela (no que diz respeito à implementação);
- *PNML2VHDL* – é uma ferramenta que permite a geração automática de código VHDL (implementação síncrona) a partir da representação PNML de um modelo RdP IOPT; e
- *PNML2C* – é uma aplicação que gera automaticamente o código ANSI C com origem na representação PNML.

2.3 Estado da arte das Networks-on-Chip

Tal como referido em [14], a ideia fundamental numa *Network-on-Chip* é aplicar a abordagem de camadas, comum em sistemas de telecomunicações e em redes de computadores. O modelo de referência OSI (*Open Systems Initiative*) não é necessariamente seguido com rigor, sendo normalmente acomodado. A rede pode ser orientada à ligação ou não orientada à ligação . É sempre necessária uma interface própria para formar o pacote e / ou estabelecer a ligação. A NoC é composta por nós (também designados comutadores) que encaminham o tráfego, podendo conter *buffers*. A topologia utilizada influencia a forma como os recursos da rede se interligarão. As propostas de topologias existentes são variadas, incluindo

topologias de interligação ad-hoc, em malha (*mesh*), em malha toroidal, em anel bidireccional, octogonais ou em árvore [14].

Em [14] apresentam-se algumas das propostas mais usuais para NoCs, das quais se realçam as seguintes:

a) *xPipes* [15] – uma rede flexível constituída pela parametrização de componentes sintetizáveis. A topologia pode ser especificada pelo projectista. Recorre-se a comutação com tabelas de encaminhamento estáticas. A rede é síncrona e, quer os comutadores, quer as ligações, são *pipelined,* de modo a obter uma elevada taxa de transmissão. Um pacote de confirmação de recepção é devolvido ao transmissor após sucesso da respectiva transmissão;
b) *Proteo* [16] – é semelhante à *xPipes* em muitos aspectos. As implementações são baseadas em interligações parametrizadas de blocos IP e a topologia pode ser seleccionada pelo projectista. Existe a opção de utilização de pacotes de confirmação, mas a implementação da retransmissão do pacote ou a correcção de erros estão a cargo do projectista;
c) *Nostrum* [17] – utiliza a topologia "clássica" em malha, com os recursos colocados nos núcleos ligados por uma matriz de interligação. O modelo OSI é utilizado, mas só as 3 camadas mais baixas (física, lógica e camada de rede) são obrigatórias. É possível formar um circuito virtual para que uma fracção do total da largura de banda disponível possa ser alocada a serviços prioritários (implementado com base numa estratégia *Time-Division Multiple Access* (TDMA) modificada);
d) *SoCbus* [18] – SoCbus (NoC circuito-comutado) utiliza uma topologia em malha 2D. O objectivo é substituir os barramentos embutidos por uma rede de comutação de circuitos para fornecer maior largura de banda. Há um único nó de coordenação - *Central Coordination Node* (CNN) - executando as funções de coordenação do sistema. O CNN gera a configuração de cada nó quando um circuito é inicializado. Não existe controlo de fluxo garantido, mas existe um sinal de confirmação;
e) *SPIN fat tree* [19] – Ao contrário de outras NoCs, a *Scalable Programmable Integrated Network* (SPIN) tem como topologia de referência uma "*fat-tree*". Trata-se de uma estrutura em árvore com

encaminhadores nos nós e recursos computacionais nas folhas, excepto nos nós com “pais” repetidos. Não considera detecções de erros ou retransmissões;

f) *XGFT* [20] – As *Generalized Extended Fat Trees* (XGFTs) podem ser construídas, com base em nós que encaminham os pacotes no sentido ascendente e no sentido descendente, com blocos de comutação XGFT separados. Foi provado que o desempenho da solução XGFT é superior ao das soluções em malha;

g) *Redes CDMA* [21] – O *Code-Division Multiple Access* (CDMA) é uma técnica bem conhecida, utilizada nas comunicações sem fios *spread-spectrum*, também sendo aplicada a barramentos embutidos. É necessário um árbitro centralizado para configurar a interface de destino de forma a receber o código do canal desejado; e

h) *Philips Æthereal* [22] – utiliza encaminhamento sem contenção, ou *pipelined time-division-multiplexed circuit switching*, para implementar os seus serviços de desempenho garantido. Embora todas as *streams* de dados tenham a mesma prioridade, podem todavia utilizar reservas de largura de banda diferentes. Os *slots* de tempo e o encaminhamento são "programados" usando tabelas residentes em cada nó. Há dois modelos de programação: distribuídos e centralizados.

O grande número de parâmetros de uma NoC (definição da topologia, comutação, política de controlo de fluxo, algoritmo de encaminhamento, etc.) é um problema inerente não só ao design, como à sua comparação [23].

Generalizando, não existe nenhuma NoC ideal. No entanto, os cenários de aplicação permitem identificar quais os parâmetros mais importantes para fazer uma análise comparativa. Em média, segundo o estudo realizado em [23], as métricas mais frequentemente e importantes são: (1) o tempo de execução; (2) a área de silício; (3) o consumo de energia; e (4) a latência. Naturalmente, estas propriedades devem ser minimizadas.

Contudo, e tendo em conta que o objectivo é o de caracterizar uma NoC que possa vir a ser gerada de forma automática por um lado, e o de comunicar entre plataformas distintas por outro, a análise do estado da arte para este trabalho deve focar-se em duas métricas:

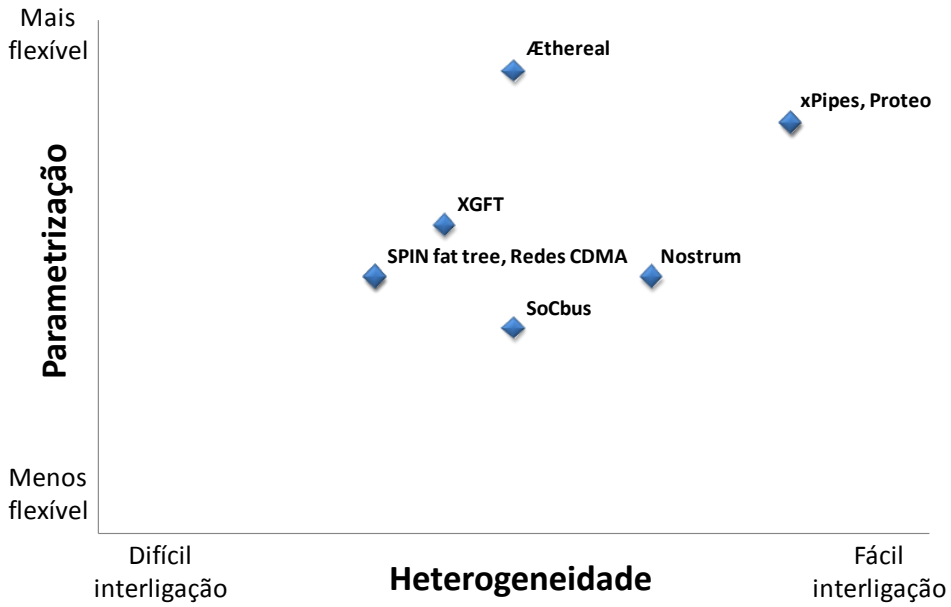


Figura 2.3 - Enquadramento das arquitecturas NoC

a) *Facilidade de interligação entre plataformas heterogéneas* – será avaliado o hardware / software necessário, bem como a forma como os dados são transmitidos. Pretende-se saber se a arquitectura em questão se adequa e / ou é flexível aos requisitos que cada plataforma tem.

b) *Flexibilidade da parametrização* – com o intuito de, através dos vários componentes, se poder vir a gerar a rede, os parâmetros que definem uma NoC não devem ser demasiado rígidos para que não venha a ser necessário o projectista intervir de forma "manual".

Apesar de não se estar perante um *benchmarking* comum, foi analisada a literatura, [15-22], e construído um gráfico, apresentado na Figura 2.3, para melhor percepção do enquadramento de cada arquitectura. O eixo da parametrização foi obtido / comparado com o estudo realizado em [24]. No que diz respeito à heterogeneidade, foi analisada a interface dos vários tipos de NoC, com a ajuda do estudo já feito em [23], permitindo estudar como os nós comunicam entre si (na perspectiva da camada física do modelo OSI). Como se pretendem plataformas heterogéneas, foi verificada a facilidade de se poder implementar a solução em hardware e software (com um peso de 50%, tendo como valor máximo quando a solução dá para ambas as plataformas e não utiliza muitos recursos) e o número de cabos necessários para efectuar a ligação (outros 50%, sendo o máximo quando não são necessários muitos cabos).

2.4 Comunicações Série

A comunicação de dados é a transmissão e recepção de informação entre dois equipamentos, através de um canal de comunicação. São transmitidos sinais (eléctricos, electromagnéticos, ópticos) que correspondem a símbolos, representando a informação digital.

Tal como já foi referido várias vezes, é necessário interligar várias plataformas heterogéneas. Uma vez que na grande maioria das plataformas está presente uma interface série de comunicação, vale a pena fazer uma breve descrição de alguns conceitos sobre este tipo de comunicação.

Numa ligação série, os dados são enviados bit a bit na via de transmissão. Contudo, já que a maior parte dos processadores trata as informações através de registos com mais do que um bit, é necessário transformar os dados (vindos dos registos) em dados série, no caso de se tratar do emissor, e de forma inversa caso se trate do receptor. Estas operações são realizadas graças a um controlador de comunicação. Generalizando, o controlador de comunicação funciona do seguinte modo:

a) A transformação paralelo-série faz-se graças a um registo de deslocamento. O registo de deslocamento permite deslocar o conteúdo do registo (o conjunto dos dados presentes em paralelo) para uma posição à direita, e seguidamente emitir o bit mais significativo e assim sucessivamente (Figura 2.4 retirada em Abril de 2010 de http://pt.kioskea.net/contents/transmission/transmode.php3)

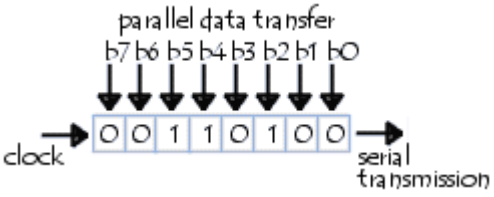


Figura 2.4 - Transformação paralelo-série.

b) A transformação série-paralelo faz-se quase da mesma maneira, utilizando um registo de deslocamento. O registo de deslocamento permite deslocar o registo de uma posição para a direita a cada recepção de um bit, e em seguida emitir a totalidade do registo em paralelo quando este está cheio e assim sucessivamente (Figura 2.5

retirada de http://pt.kioskea.net/contents/transmission/transmode.php3 em Abril de 2010)

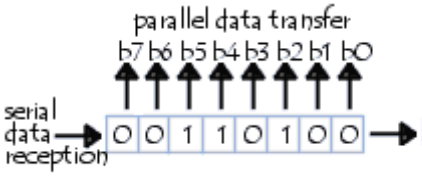


Figura 2.5 - Transformação série-paralelo.

Tal como Fréderic Leens disse em [25], quando há a necessidade de implementar uma comunicação entre um circuito integrado, como um microcontrolador, e um conjunto de periféricos relativamente lentos, e / ou com pouco espaço de memória, não há vantagem em usar protocolos excessivamente complexos. Nestas situações, um engenheiro de sistemas embutidos prefere protocolos como o I^2C, SPI ou, neste caso RS-232, que cumprem perfeitamente os requisitos e são populares [25]. No "mundo" dos protocolos de comunicação, I^2C, SPI e RS-232 são muitas vezes considerados como "pequenos" protocolos de comunicação em relação ao Ethernet, USB, SATA, PCI-Express e outros. Mas é importante não esquecer a finalidade de cada protocolo.

2.4.1 Protocolo RS-232 (EIA232)

A especificação RS-232 define as características de mecanismo, eléctricas e funcionais. RS-232 é uma interface *single ended*, unidireccional (ponto-a-ponto) - o sinal é referenciado à massa. Os *drivers* RS-232 adaptam a tensão do sinal RS-232 (-15 V a 15 V) para níveis usados na lógica TTL (0 V a 5 V).

A interface RS-232 está classificada para operar até 20kbps. Apesar do comprimento máximo do cabo não estar definido, é delimitada uma capacidade máxima da linha de 2500pF, com uma carga de impedância entre 3KΩ e 7KΩ. Isso origina um comprimento máximo de aproximadamente 20 metros [26].

Normalmente, os dados são enviados em 7 ou 8 bits. Um bit de início (*start bit*) assinala o início da trama. O *start bit* é activo com o valor lógico "zero" (o *driver* RS-232 inverte a sinalização, por isso é activo a "um"). A Figura 2.6 (retirada em Abril de 2010 de http://www.interfacebus.com/Design_Connector_RS232.html) mostra uma trama de 8 bits (antes da inversão). Após o *start bit* seguem-se os dados. O valor lógico "um" é representado com uma tensão negativa entre -3V e -15V. Dependendo

do protocolo utilizado, pode existir um bit de paridade. Normalmente também existe um bit para sinalizar o fim da trama (*stop bit*).

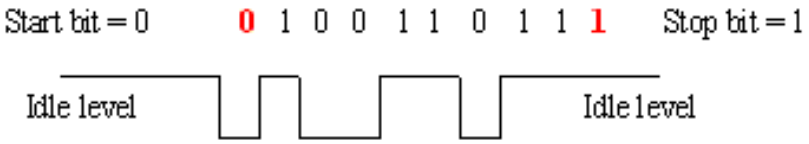
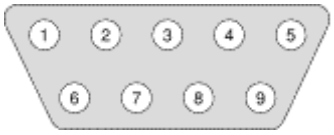


Figura 2.6 - Trama RS-232

A especificação RS-232 define apenas o pin-out para o conector D-sub de 25 pinos. No entanto, o mais utilizado é o conector de 9 pinos (definido pela EIA-574).



Pino #	***Nome do Sinal***	***Descrição do sinal***
1	CD	Carrier detect
2	RXD	Receive data
3	TXD	Transmit data
4	DTR	Data terminal ready
5	GND	Ground
6	DSR	Data set ready
7	RTS	Request to send
8	CTS	Clear to send
9	RI	Ring indicator

Figura 2.7 - Conector RS232 DB-9 pin-out

2.4.2 SPI – Serial Peripheral Interface

O SPI é um protocolo de comunicação *single-master*, isto é, um dispositivo central (*master*) que inicia todas as comunicações com os *slaves*. Este é bastante simples, definido, baseado em quatro linhas de sinal, como é demonstrado na Figura 2.8, retirada de [25] :

a) Um sinal de relógio (SCLK) enviado pelo *master*, a partir de um barramento, para todos os *slaves*;
b) Um sinal por cada *slave* (SS) para seleccionar qual o *slave* com que o *master* se comunica;
c) Linha de dados do *master* para o *slave* (MOSI); e
d) Linha de dados do *slave* para o *master* (MISO).

Quando o *master* pretende enviar dados e/ou solicitar informações, selecciona um *slave*, activando a linha SS correspondente, e activa o sinal de relógio com uma frequência de relógio que possa ser usada tanto pelo *master* como pelo *slave*. O *master* transmite informações na linha MOSI e lê na linha MISO (Figura 2.9 [25])

O SPI não define qualquer taxa máxima de dados nem qualquer esquema de endereçamento específico; não tem mecanismo de reconhecimento para confirmar a

recepção dos dados e não oferece qualquer controle de fluxo. Na realidade, o *master* nem tem conhecimento se existe algum *slave*, a não ser que algo adicional seja feito "extra" protocolo. Também não se preocupa com as características físicas, como a interface de entrada/saída e tensões padrão usadas entre os dispositivos.

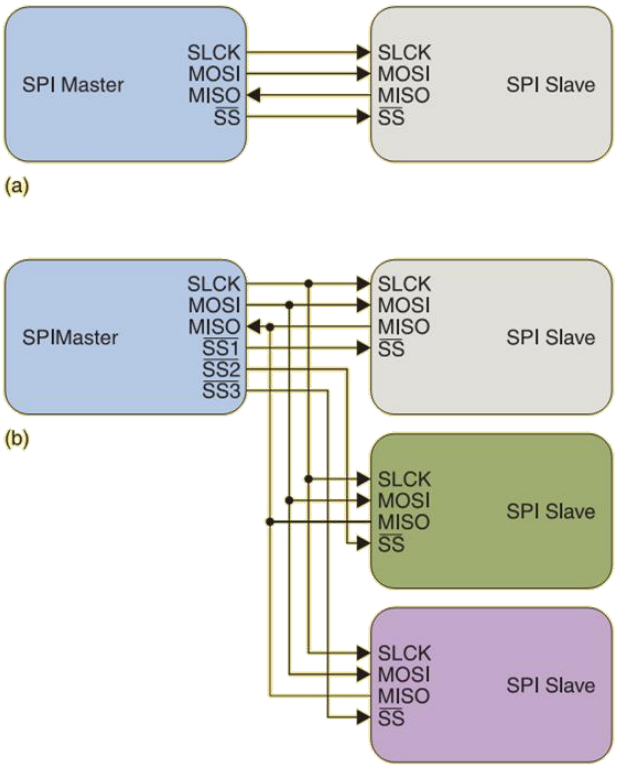


Figura 2.8 - Duas topologias SPI [25]. (a) master SPI ligado a um único slave. (b) master SPI ligado a vários slaves.

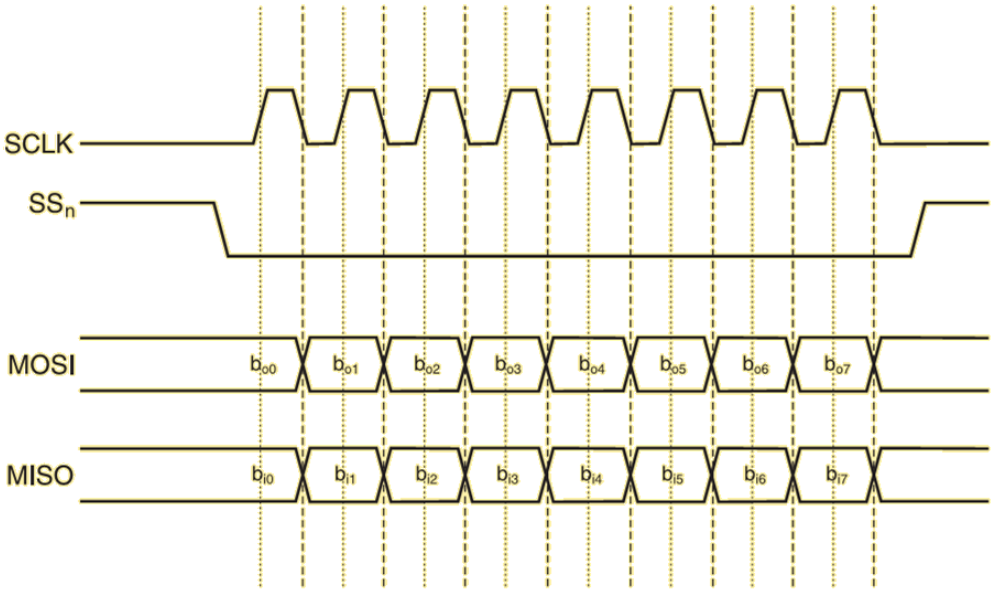


Figura 2.9 - A comunicação SPI simples [25]. Transmissão de dados em MOSI e resposta em MISO no flanco ascendente do sinal de relógio SCLK.

2.4.3 I²C - Inter-Integrated Circuit

O I²C é um protocolo *multi-master* que utiliza duas linhas de sinais, o *serial data* (SDA) e o *serial clock* (SCL), para os sinais de dados e o relógio, respectivamente. Não existe a necessidade de seleccionar qual o chip (*slave*) nem é necessária lógica de arbitragem.

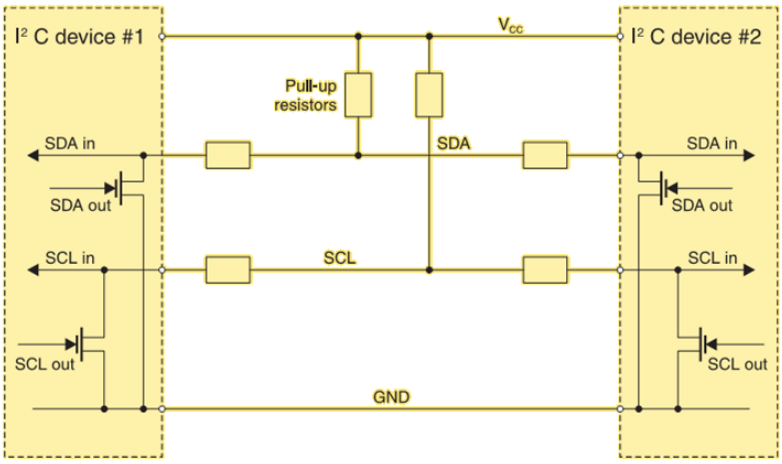


Figura 2.10 - Barramento I2C com dois dispositivos conectados [25].

A Figura 2.10, retirada de [25], mostra como dois dispositivos são ligados. Tanto a linha SCL como a SDA são entradas/saídas *open-drain* com resistências de *pull-up*. Caso se pretenda o valor lógico ZERO coloca-se a linha a *ground*, ao passo que se a linha não tiver nada imposto é forçada a tomar o valor lógico UM.

De modo geral, inúmeros *slaves* e *masters* podem ser ligados a estas duas linhas de sinais para se comunicarem, utilizando o protocolo que define:

- Endereço de 7 bits – cada dispositivo conectado ao barramento possui um endereço único;
- Os dados são divididos em bytes; e
- Alguns bits de controlo para gerir o início, o fim, a direcção da comunicação e o mecanismo de reconhecimento.

A especificação do protocolo I²C refere que o dispositivo que começa a transferência de informação é o master do barramento. Consequentemente, até esse momento todos os dispositivos são considerados slaves. Primeiro, o master irá emitir a condição COMEÇAR. É como que um sinal de “atenção” para todos os dispositivos conectados. Todos irão estar atentos ao barramento e analisar os dados de entrada. Depois, o master envia o endereço de quem quer aceder, com a informação sobre se o acesso é uma operação de escrita ou de leitura. Todos os

dispositivos comparam o endereço recebido com o seu. Se não coincidir, libertam o barramento até à condição de PARAR. Se coincidir, o dispositivo produz uma resposta de RECONHECIMENTO. Os dados são então transmitidos. Quando a operação estiver concluída o master emite a condição de PARAR, que também sinaliza que o barramento está livre (Figura 2.11). A Tabela 2.1 demonstra o pacote desta descrição.

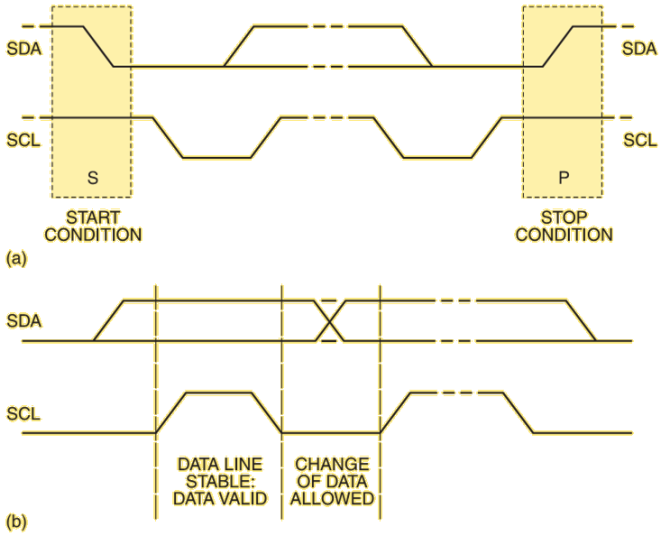


Figura 2.11 - Quando os dados são transmitidos no barramento I2C [25]. (a) Mostra o sinal de dados SDA relativamente à fase do sinal de relógio SCL. A alteração de dados quando SCL não é estável e no nível lógico "zero" não é permitida. (b) mostra as condições de COMEÇAR

Tabela 2.1 - Pacote I^2C [25].

Começar	Endereço do *slave*	$RD/\overline{WR}$	Rec.	Dados	Rec.	…	Dados	Rec.	Parar
1 bit	7 bits	1 bit	1 bit	8 bits	1 bit	…	8 bits	1 bit	1 bit

Associando a camada física e o protocolo descrito, é permitida a comunicação entre dispositivos em apenas dois fios sem falhas. Segundo [25], o I^2C é extremamente elegante neste aspecto.

Por exemplo, se dois dispositivos colocarem ao mesmo tempo informações sobre o SDA e o SCL não existe conflito eléctrico. O ZERO lógico imposto "ganha" sempre e nunca existe curto-circuito graças à estrutura de *pull-up*. Fisicamente é também possível escrever e ler ao mesmo tempo. Dessa forma, qualquer dispositivo é capaz de detectar colisões (um dispositivo coloca o valor lógico UM e outro ZERO). Como para existir uma colisão é necessário terem colocado dados

diferentes, e só o que colocou o valor UM é que se apercebe, este desiste; e o outro *master* mantém a comunicação não dando pelo sucedido.

2.5 Comunicação entre SoCs

O facto de existir um sistema dividido em subsistemas e de estes poderem vir a ser implementados em sistemas heterogéneos, faz com que se deva ter em consideração que, estes sistemas podem ser localmente síncronos e globalmente assíncronos (*Globally Asynchronous and Locally Synchronous* - GALS). Ou seja, os componentes são executados de forma concorrente, não sendo certo que tenham o mesmo ritmo de processamento e que a comunicação entre eles seja feita de forma síncrona.

Alain J. Martin e Mika Nyström descrevem em [27] algumas técnicas para o design deste tipo de sistemas. Considerando a mesma nomenclatura, estes componentes são vistos como processos. Os processos não partilham “variáveis” e apenas comunicam através do envio e recepção de “acções” [27]. Os dados transferidos através de um canal, durante uma comunicação, são denominados *mensagem*. A implementação da comunicação *enviar / receber* é crucial para os métodos da lógica assíncrona bem como para o objectivo deste trabalho, uma vez que se pretende que a solução seja transparente para o projectista. Ou seja, este só tem que se preocupar com os modelos e não com a forma como eles se interligarão.

Normalmente, e de modo não transparente, utiliza-se o protocolo *handshake*. O *handshake* é utilizado em protocolos como o FTP, TCP, HTTP, SMTP, POP3, entre outros. O seu funcionamento é extremamente simples. Existem dois canais de comunicação, um em cada sentido. Num deles é transmitido um sinal instruindo para o processo B “actuar”, sendo devolvido ao processo A através do outro canal, quando este tiver conferido a informação “actuar”.

2.6 Buffers

O *buffer* é uma região de memória temporária para a escrita e leitura de dados. Um *buffer* comporta-se como um armazém, possui *slots* (lugares) para guardar informação (objectos), sendo estes *slots* identificados por índices. Os *buffers* podem ser implementados, quer em *software,* quer em *hardware,* e são normalmente utilizados quando existe uma diferença entre a velocidade a que a informação é recebida e a velocidade a que ela pode ser processada.

Tendo em conta que se quer ter uma interface transparente com o modelo, verifica-se que será necessário um *buffer* (em vez dum protocolo *handshake*), para que não se perca nenhuma informação na comunicação.

Será utilizado um *buffer* FIFO (*First In – First Out*), ou seja, a informação é processada pela ordem de chegada. Existem várias maneiras de implementar um *buffer* FIFO das quais se destacam:

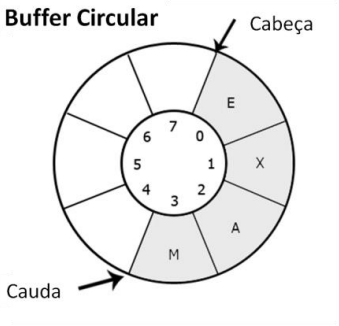


Figura 2.12 - *Buffer* circular

a) ***Buffer* circular** (Figura 2.12)

Tipicamente são necessários dois apontadores para o seu funcionamento (Figura 2.13): (1) um para apontar para o início dos dados (válidos) – *apontador de leitura*; e (2) outro para apontar para o fim dos dados (válidos) – *apontador de escrita*.

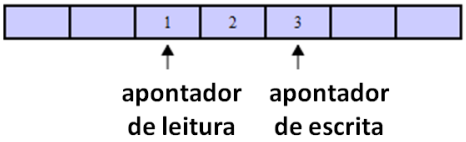


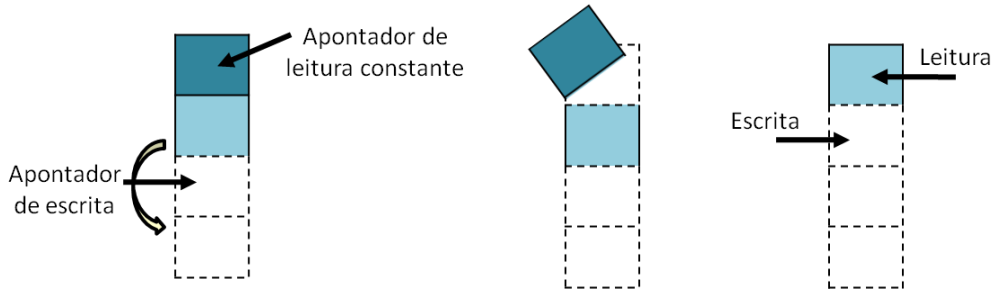
Figura 2.13 - Apontadores do *buffer* circular

Este tipo de *buffer* tem o problema da distinção entre o *buffer* cheio e o *buffer* vazio, quando os dois apontadores estão na mesma posição. Uma das soluções é deixar sempre uma *slot* vazia, a outra é ter um contador auxiliar.

b) ***Buffer* baseado em Registos de deslocamento** (Figura 2.14)

Tal como o *buffer* circular, o seu funcionamento baseia-se em apontadores. Existe um apontador que indica em que posição do *buffer* devem ser escritos os dados, e a leitura é sempre feita na cabeça da lista. Cada vez que ocorre a leitura, o elemento da cabeça da lista é

retirado e todos os outros são deslocados nessa direcção, assim como o apontador de escrita.



a) Apontadores **b) Processo de leitura de um elemento**

Figura 2.14 - *Buffer* baseado em Registo de deslocamento

2.7 Ferramentas relacionadas

Como referido, o objectivo deste trabalho é o de apresentar uma solução, aproveitando-se de soluções do tipo *Network-on-Chip*, para a interligação de componentes, provenientes de modelos RdP IOPT, que possa vir a ser obtida através de um gerador. Não foi encontrada nenhuma ferramenta que já o fizesse.

João Oliveira, em [28], tem como objectivo implementar uma ferramenta que permita a geração de ficheiros de configuração adequados a plataformas específicas, e especificados através de modelos de RdP, num contexto de co-design de sistemas embutidos, possibilitando a integração de um conjunto de componentes previamente definidos. No seu trabalho de pesquisa, conclui que as ferramentas relacionadas com o seu trabalho não cumpriam os objectivos e que uma nova ferramenta teria que ser realizada, concebendo um protótipo constituído pelo: *PDC Editor*; e *PDC Generator*.

Tendo como objectivo obter uma solução que possua uma elevada flexibilidade de interligação entre componentes em plataformas heterogéneas, para tal baseada numa rede de comunicação série, este protótipo não preenche os requisitos. Limita-se a integrar os componentes de forma *ponto-a-ponto* entre as plataformas, sendo necessário criar cablagem específica.

3 Descrição da solução proposta

A primeira secção deste capítulo expõe o ambiente de desenvolvimento para que na segunda secção se possa fazer um enquadramento da solução. Ainda neste capítulo, nas restantes secções, serão apresentadas as regras e alguns *templates* para que, quando conjugados, possam formar a rede.

3.1 Ambiente de desenvolvimento

Tal como já foi mencionado, pretende-se com este trabalho apresentar uma solução de interligação de componentes previamente gerados a partir de modelos expressos em RdP IOPT. A solução será codificada em VHDL e / ou em C conforme a plataforma de implementação.

Os componentes referidos serão obtidos como resultado da partição de um modelo expresso em RdP, de acordo com as regras de partição propostas em [29], realizada utilizando o editor de RdP SNOOPY-IOPT [4] em conjugação com a ferramenta SPLIT [4] e com as ferramentas de geração automática de código C e VHDL a partir das representações PNML dos modelos RdP resultantes da partição.

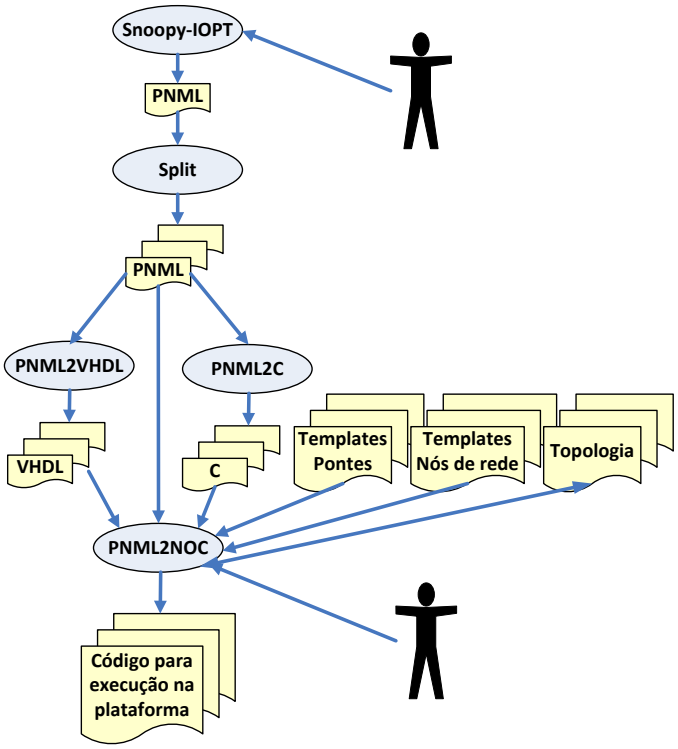


Figura 3.1 - Fluxo de desenvolvimento e ferramentas de suporte.

Para o desenvolvimento da solução proposta serão consideradas as ferramentas atrás referidas, desenvolvidas no âmbito do projecto FORDESIGN onde

se identifica a interligação de uma nova ferramenta, designada por PNML2NOC, a desenvolver em trabalho futuro, resultando das regras propostas ao longo desta dissertação (Figura 3.1). O SNOOPY-IOPT permite a edição de redes de Petri IOPT e posterior representação num ficheiro PNML. As ferramentas PNML2C e PNML2VHDL geram automaticamente o código C e VHDL, respectivamente, a partir dos ficheiros PNML obtidos pela ferramenta SPLIT.

A selecção do PNML2C e/ou do PNML2VHDL para a geração do código associado a um determinado componente está relacionada com as plataformas escolhidas para a execução desses componentes; os *templates* que se irão utilizar para produzir os nós de interligação através da NoC também são afectados por essa escolha. Estes últimos são ficheiros VHDL e C que serão alterados conforme as definições dos nós da rede. Estas definições estarão num ficheiro (utilizando formato XML) e ditam qual a topologia, a posição de cada componente e as velocidades na rede.

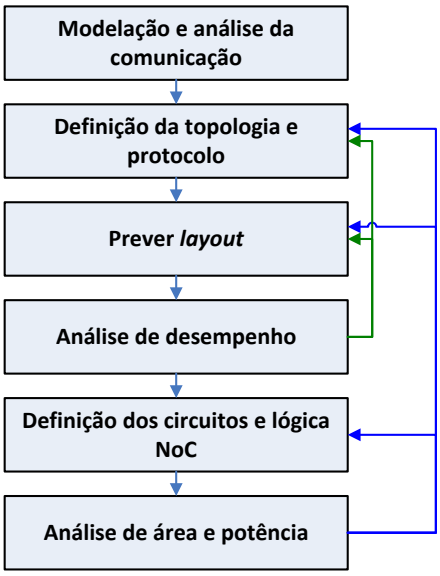


Figura 3.2 - Metodologia de construção de uma NoC [30]

A metodologia para definir as regras propostas terá como base o diagrama da Figura 3.2 proposto em [30], embora não tenha sido seguido de forma muito rigorosa. O primeiro passo será a modelação da comunicação e a sua análise. O segundo passo consiste em definir a topologia e o design do protocolo baseado nas exigências de comunicação. O seu objectivo é usar o mínimo de recursos da rede, para cumprir as exigências de comunicação apontada pelo modelo de comunicação. No terceiro passo deve prever-se o atraso em cada ligação, em termos de ciclos de relógio, e analisar a potência e a área utilizada para, que se cumpram algumas

exigências do projecto de ligações, porque na arquitectura NoC as ligações consomem potência e área. O quarto passo é a análise de desempenho. Os resultados de desempenho ajudam a comparar escolhas de arquitectura diferentes e a refiná-las. No quinto passo, a arquitectura lógica pode ajudar a aumentar a exactidão da análise de potência e área. O último passo é a análise de área e potência. Se a potência ou a área não satisfizerem as condições, volta-se a fases anteriores de modo a escolher diferentes circuitos ou redesenhar o *layout*. No pior dos casos, a topologia e o protocolo têm que ser redesenhados.

Um dos problemas encontrados foi o de não existirem trabalhos similares, que sejam do domínio público, e que partam de componentes expressos em PNML. Optar-se-á por uma topologia em anel tendo em conta que se pretende flexibilidade e simplicidade na solução adoptada para interligação. É uma topologia que, apesar de poder aumentar o tempo de transmissão, tem um custo reduzido de recursos (hardware e software).

Para a interligação entre plataformas a escolha recai no protocolo RS-232 (podendo, embora, operar a velocidades muito superiores). Desta forma, incluem-se as soluções de hardware reconfigurável, os microcontroladores de baixo custo, habitualmente utilizados em tarefas de controlo, bem como a interligação a outros sistemas computacionais possuidores de interfaces normalizados do tipo RS-232. A selecção de um suporte à comunicação com as características do RS-232 tem várias vantagens, de entre as quais se realçam: (1) estar disponível na maior parte dos dispositivos controladores; (2) permitir uma fácil ligação; (3) ser de baixo custo; e (4) permitir a integração com sistemas distribuídos de automação, nomeadamente com os equipamentos associados, como os sistemas de manufactura e os de controlo predial. Como desvantagem principal deve ser referido o ritmo de transferência de dados relativamente baixo, quando comparado com outras soluções. No entanto, considera-se que a flexibilidade descrita, permitindo interligar dispositivos intra-circuito e inter-circuitos, compensa as limitações expostas.

Dada a escolha de interligação entre plataformas e do estado da arte das NoCs (Figura 2.3), optou-se por uma solução baseada em Networks-on-Chip Proteo. Isto é, ao longo deste capítulo apresentar-se-ão as regras e módulos que permitirão vir a gerar de forma automática, a partir das RdP IOPT representadas em PNML, uma rede similar às actuais redes Proteo.

Para futuramente se poder melhorar a solução que se irá propor, a postura a adoptar será modular. Ou seja, a solução será composta por vários módulos que comunicam entre si. Como nem sempre estes módulos estão sincronizados entre si, utilizar-se-á um protocolo *handshake.* Este método só poderá ser utilizado na comunicação entre processos da solução que não comuniquem directamente com os componentes gerados. Caso estes não possuam os canais de comunicação necessários ao protocolo, será adoptado um *buffer*.

O *buffer* que melhor se aplica é um *buffer* FIFO baseado em registos de deslocamento, por ser fácil de se implementar em *hardware* e por não se desperdiçar nenhuma posição de memória. Será descrito em VHDL com as variáveis genéricas *Buffer_Lenght* e *Vector_Width* para que se possa alterar a respectiva profundidade e comprimento.

3.2 Enquadramento da solução

Para a geração automática de redes que interliguem componentes, será necessário criar uma aplicação que implemente um conjunto de regras.

A Figura 3.3 mostra o enquadramento destas definições e como deverá ser estruturada a aplicação (ainda de forma abstracta) na arquitectura ICE. ICE (Interface, Controlo e Entidade) é uma metodologia clássica, especificada em [31], usada na arquitectura de software orientada à programação por objectos. Consiste em organizar os objectos pelas três camadas que compõem a metodologia. Usando esta metodologia, pode-se então descrever o sistema do gerador como:

- ***Interface***: É responsável por interagir com os actores do sistema. Isto é, é composta pelas classes que interagem com:
  - O projectista, de forma a obter informação sobre:
    - Quais os ficheiros PNML que geraram os módulos a interligar; e
    - Quais as plataformas onde serão implementados esses módulos, nomeadamente velocidades do relógio.
  - Os ficheiros PNML para extrair todos os sinais e eventos que comunicam com o exterior do módulo.
- ***Controlo***: Tendo toda a informação necessária, é responsável por:
  - Combinar, através da sua designação, os sinais / eventos de saída com os de entrada;

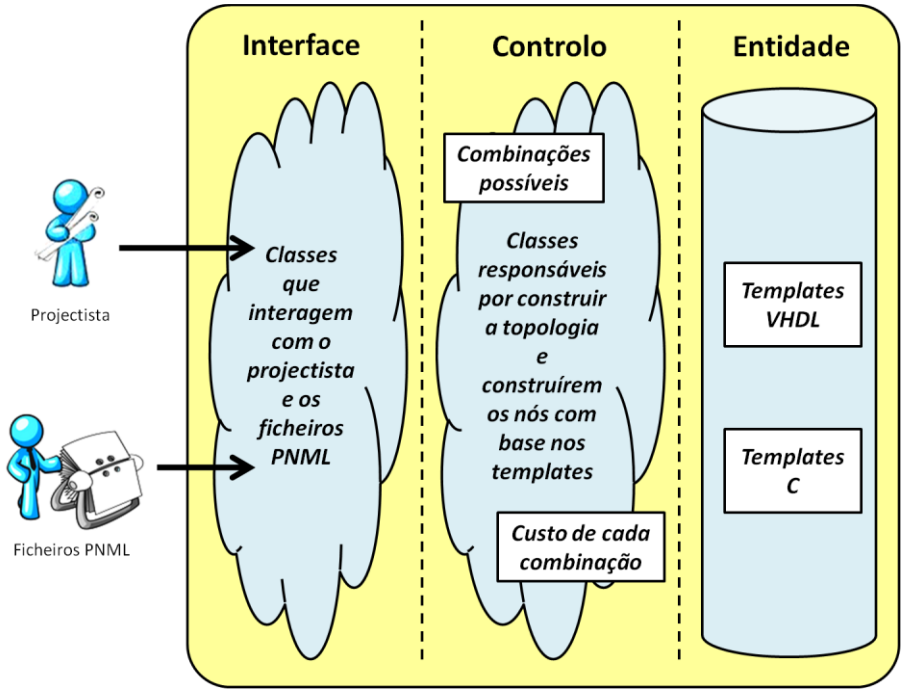


Figura 3.3 - Enquadramento dos módulos e regras definidos na arquitectura ICE

- Atribuir-lhes uma codificação;
- Em função do número de módulos a unir, criar todas as combinações possíveis de interligá-los com o respectivo custo;
- Escolher a melhor combinação de forma a atribuir uma codificação a cada nó;
- A partir das codificações obtidas, cria o protótipo das mensagens que circularão na rede, ou seja, terá um conjunto de variáveis que comporão as mensagens; e
- Com todos esses dados, interagirá com a camada *entidade* para, com base nos *templates*, construir os nós pretendidos.

- ***Entidade***: É a camada responsável pelo repositório de todos os *templates* sugeridos. É uma biblioteca onde estão contidos todos os ficheiros VHDL e C pelos quais os nós são compostos. Ou seja, combinando todos estes ficheiros, e alterando pequenos excertos, ter-se-á o resultado final.

As regras, que serão apresentadas na segunda e terceira secção, e os *templates* sugeridos, expostos nas restantes secções, estão caracterizados por rectângulos brancos na Figura 3.3. De forma pouco precisa, poder-se-á dizer que as regras apresentadas serão alguns dos algoritmos que ficarão nas classes da camada de controlo e os *templates* serão ficheiros que estarão na camada de

entidade. As restantes classes, ainda por implementar, estão representadas por nuvens.

3.3 Topologia

É utilizada uma topologia em anel (*daisy-chain*) para garantir a interligação, onde cada módulo possui um componente (associado a um submodelo gerado pela partição da RdP) e um nó de rede, através do qual se ligará à rede. No caso de existirem plataformas heterogéneas existe um nó especial com a designação de ponte. A Figura 3.4 ilustra a interligação entre módulos.

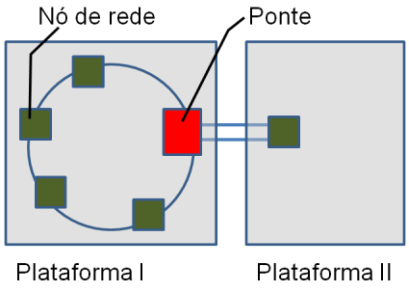


Figura 3.4 - Arquitectura de ligação entre nós em plataformas distintas utilizando uma ponte.

Na subsecção seguinte apresentam-se as regras propostas para atribuição dos endereços aos vários nós de rede.

3.3.1 Endereço de rede / posição na topologia

Cada nó tem o seu próprio endereço de rede, bem como a sua posição na topologia, que são definidos tendo em conta o número de mensagens enviadas/recebidas, o número de saltos e o ritmo de transmissão da rede.

Tal como é exposto atrás, e ilustrado na Figura 3.1, o utilizador começa por modelar o sistema, ficando numa fase posterior, com o mesmo número de ficheiros PNML que (sub)modelos. Nestes ficheiros estão caracterizados os sinais e os eventos de entrada e de saída. Cada vez que um sinal ou evento de um módulo (até então designado por modelo) comunica com o sinal de outro módulo, equivale a uma mensagem a circular na rede. A mensagem só é enviada quando esse sinal mudar de valor, ou no caso de ocorrer um evento.

Com o objectivo de reduzir a latência na rede terá que ser analisado o número de mensagens que os módulos trocam entre si e a velocidade a que o conseguem fazer (Tabela 3.1). Tal como indicado, o número de mensagens pode ser estimado

com recurso aos ficheiros PNML. O ficheiro não é mais que um XML, subtipo de SGML (*Standard Generalized Markup Language*) capaz de descrever diversos tipos de dados [32]. Pela sua portabilidade, uma vez que não depende das plataformas, quer de hardware, quer de software, o conjunto de informação pode, através de uma aplicação, ser lido dum ficheiro XML. Em suma, como não se está a admitir o conhecimento interno do modelo mas apenas a sua interface, o número de mensagens geradas entre dois (sub)modelos é estimado pelo número de interligações entre esses módulos, independentemente do número de mensagens efectivamente geradas. As velocidades terão que ser dadas pelo utilizador, especificando as plataformas e suas características, onde ficarão os módulos.

Tabela 3.1 - Análise do número de mensagens e ritmo de transmissão entre modelos.

Número de mensagens

Origem \ Destino	Nó 1	Nó 2	…	Nó N
Nó 1	-	$\#Msgs_{1\rightarrow2}$	...	$\#Msgs_{1\rightarrow N}$
Nó 2	$\#Msgs_{2\rightarrow1}$	-	…	$\#Msgs_{2\rightarrow N}$
…	…	…		…
Nó N	$\#Msgs_{N\rightarrow1}$	$\#Msgs_{N\rightarrow2}$	…	-

Ritmo de Transmissão [bps]

Origem \ Destino	Nó 1	Nó 2	…	Nó N
Nó 1	-	$Vel_{1\rightarrow2}$	…	$Vel_{1\rightarrow N}$
Nó 2	$Vel_{2\rightarrow1}$	-	…	$Vel_{2\rightarrow N}$
…	…	…		…
Nó N	$Vel_{N\rightarrow1}$	$Vel_{N\rightarrow2}$	…	-

Observando-se como é construída a tabela do ritmo de transmissão, verifica-se que existem dados redundantes. A velocidade de envio do Nó 1 para o Nó 2, por exemplo, não poderá ser superior à capacidade de recepção do Nó 2. O cálculo é feito com recurso a uma tabela intermédia (Tabela 3.2), obtida aquando da definição das plataformas onde serão implementados os módulos. É comparada a velocidade de envio e de recepção, e o resultado final é a menor das duas.

Tabela 3.2 - Velocidade de cada nó de rede

	Recepção	***Transmissão***
Nó 1	Vel_Rec_1	Vel_Trans_1
Nó 2	Vel_Rec_2	Vel_Trans_2
…	…	…
Nó N	Vel_Rec_N	Vel_Trans_N

Sabendo que se está perante uma topologia em anel, das tabelas anteriores (Tabela 3.1) calcula-se para todos os casos possíveis o custo que terá a posição e a disposição de cada nó no anel. Ou seja, se, por exemplo, tivermos 4 nós em que o primeiro nó só comunica com o segundo, o segundo com o terceiro e o terceiro com o quarto, empiricamente, os nós terão que estar próximos (…→Nó1→Nó2→Nó3→Nó4→ Nó1→…). Para que se possam analisar os dados de

forma automática, através de uma aplicação, ter-se-á que aplicar análise combinatória. Tal como referido em [33], com o cálculo combinatório podem-se contar diferentes modos de percorrer determinados caminhos, usando maneiras sistemáticas de proceder. É ainda de referir que, como a velocidade de transmissão pode ser diferente da velocidade de recepção, o custo de se ir do nó 2 para o nó 1 pode ser diferente do custo de se ir do nó 1 para o nó 2. Assim, está-se perante N! combinações, sendo N o número total de nós. Está-se perante uma permutação simples. Já existe bastante literatura, como em [3, 34-37], com algoritmos para encontrar todas a hipóteses de caminhos possíveis. A escolha recai no algoritmo de Heap. O algoritmo de Heap é um pequeno e elegante algoritmo implementado utilizando o método recursivo HeapPermute [3]. É invocado por HeapPermute(N), continuando N a ser o número de elementos. Na Figura 3.5 é mostrado o fluxograma do algoritmo de Heap (HeapPermute).

Como resultado do algoritmo apresentado, ter-se-á N! linhas, sendo cada linha

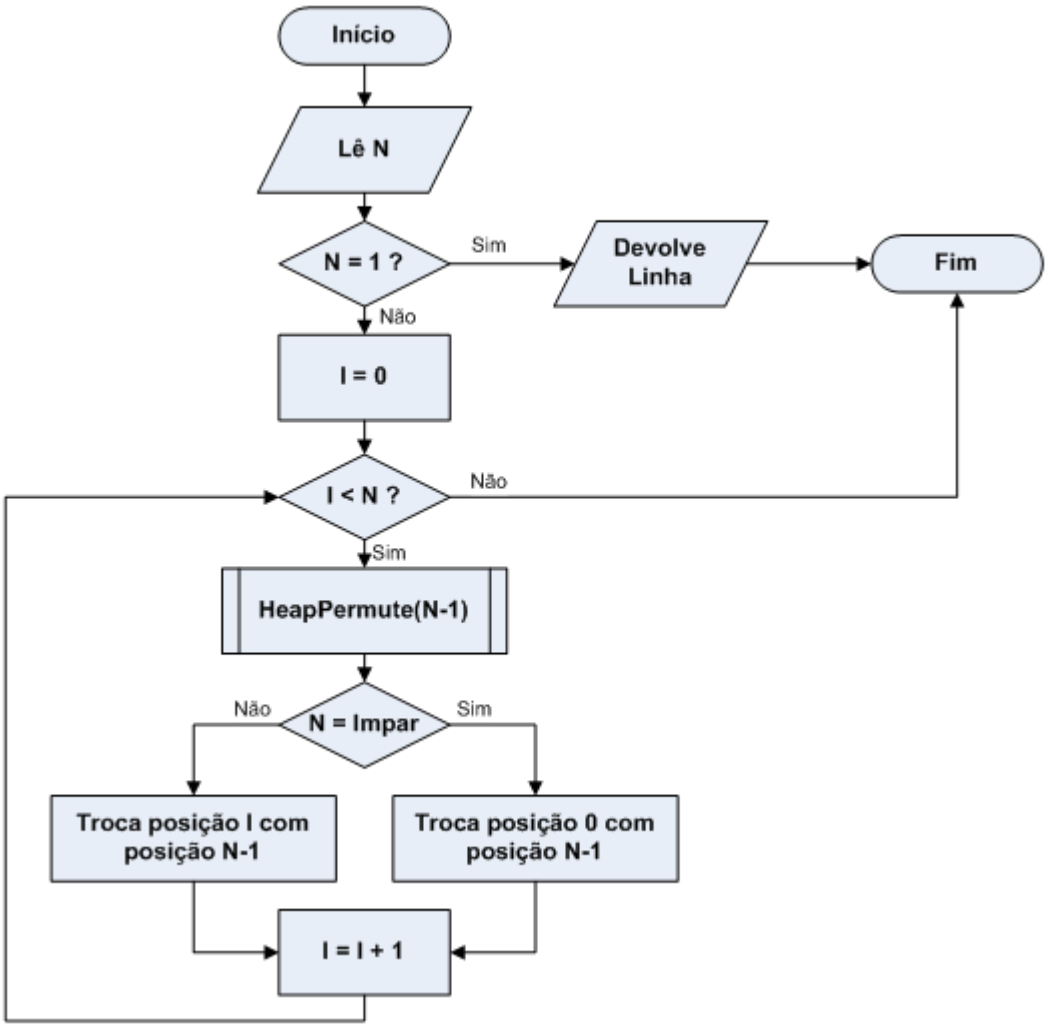


Figura 3.5 - Fluxograma do algoritmo de Heap – HeapPermute(N) [3].

uma das combinações possíveis (Tabela 3.3). O próximo passo para se saber o custo associado a cada topologia, isto é, a cada combinação de interligação dos nós (cada linha), é ter em conta o número de mensagens enviadas por cada nó, a velocidade a que o consegue fazer e quantos saltos terá que dar até chegar ao destino.

Tabela 3.3 - Demostração, a titulo de exemplo, do resultado do algorimo de Heap

	Posição 1	***Posição 2***	***…***	***Posição N***
Combinação 1	Nó1	Nó2	…	NóN
Combinação 2	Nó2	Nó1	…	NóN
…	…	…		…
Combinação N	NóN	Nó2	…	Nó1

O pseudocódigo da Figura 3.6 mostra o algoritmo utilizado, considerando as seguintes matrizes:

- $A = \langle a_{ij} \rangle_{N \times N}, em\ que\ a_{ij}$ é $\#Msgs_{i \to j}$ (Tabela 3.1) $e\ N\ o\ número\ de\ Nós$

- $B = \langle b_{ij} \rangle_{N \times N}, em\ que\ b_{ij}$ é $\frac{Vel_{i \to j}}{\max\ (Vel_{ij})}$ (Tabela 3.1) $e\ N\ o\ número\ de\ Nós$

- $C = \langle c_{ij} \rangle_{N! \times N}$, $em\ que\ c_{ij}$ é $o\ Nó\ da\ combinação\ i\ na\ posição\ j$ da Tabela 3.2$e\ N\ o\ número\ de\ Nós$

Sucintamente, este algoritmo consiste em percorrer todas as posições de cada combinação da matriz C (equivale à Tabela 3.3) analisando qual o nó correspondente. É verificado se esse nó envia mensagens (Tabela 3.1) e, em caso afirmativo, é tido em conta como um factor de custo e são apurados quais os seus destinos. Serão percorridas todas as posições até que uma delas corresponda ao nó aferido. Enquanto não for encontrada a posição correcta, será considerado o ritmo de transmissão entre nós como mais um factor.

Uma vez aplicado o algoritmo, ter-se-á o custo associado às N! combinações possíveis de interligação entre os nós. A escolha recai na que tiver menor custo. Desta forma já se sabe "onde colocar" cada módulo e atribuir-se-á um endereço a cada nó de rede. Os endereços serão codificados em binário com Y bits, sendo que $2^Y \geq N \Leftrightarrow Y \geq \log_2(N), com\ Y\ e\ N \in \mathbb{N}$. O nó da primeira posição ficará com o endereço "…00", o nó da segunda posição ficará com o endereço "…01", e assim sucessivamente.

INÍCIO
DESDE combinação = 0 **ATÉ** combinação = N! **REPETIR**
$Custo_{combinação} = 0$
DESDE posição = 1 **ATÉ** posição = N **REPETIR**
$Origem = C_{(combinação,posição)}$
DESDE destino = 1 **ATÉ** destino = N **REPETIR**
$Msgs = A_{(origem,destino)}$
SE Msgs > 0 **ENTÃO**
Posição1 = posição
Posição2 = posição + 1
SE Posição2 > N **ENTÃO**
Posição2 = 1
FIM SE
FAZER
$P1 = C_{(combinação,Posição1)}$
$P2 = C_{(combinação,Posição2)}$
$VelP1P2 = B_{(P1,P2)}$
$Custo_{combinação} += Msgs * VelP1P2$
Posição1 = Posição1 + 1
SE Posição1 > N **ENTÃO**
Posição1 = 1
FIM SE
Posição2 = Posição2 + 1
SE Posição2 > N **ENTÃO**
Posição2 = 1
FIM SE
ENQUANTO destino diferente de P2
FIM SE
destino = destino + 1
FIM DESDE
posição = posição + 1
FIM DESDE
combinação = combinação + 1
FIM DESDE
FIM

Figura 3.6 – Pseudocódigo do algoritmo utilizado para calcular o custo associado a cada combinação.

3.4 Protocolo

A nível físico ir-se-á utilizar um protocolo baseado no RS-232. Assim, nesta secção serão definidas, para os outros níveis do modelo de referência OSI, um conjunto de normas que todos os nós que intervenham na comunicação de dados terão que cumprir.

Tal como já referido no ponto anterior, há uma topologia em anel onde existe uma especificação, através de um endereço de Y bits, de forma a identificar o terminal concreto da rede com o qual se estabelecerá uma comunicação.

3.4.1 Mensagens

O protocolo baseia-se em transmissão de pacotes. Para tal, os sinais e / ou os eventos serão organizados em tramas contendo os dados de origem (para que se

possa vir a receber uma mensagem de reconhecimento) e de destino, as quais vão circulando de nó em nó até chegar ao endereço. Para além destes dois dados, cada mensagem (Figura 3.7) contém:

a) Um campo de um bit a designar se é ou não uma mensagem de reconhecimento;
b) A designação do sinal / evento contido – código que cada nó tem associado aos sinais / eventos de entrada;
c) O seu valor;
d) Um código de quatro bits para que se consiga identificar se a mensagem a circular na rede é fruto de uma falha e se esta já foi processada ou, se é uma mensagem nova do mesmo sinal com o mesmo conteúdo; e
e) Um campo de verificação de redundância cíclica (CRC – *Cyclic redundancy check*).

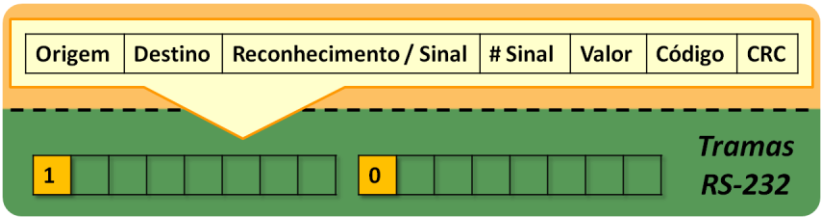
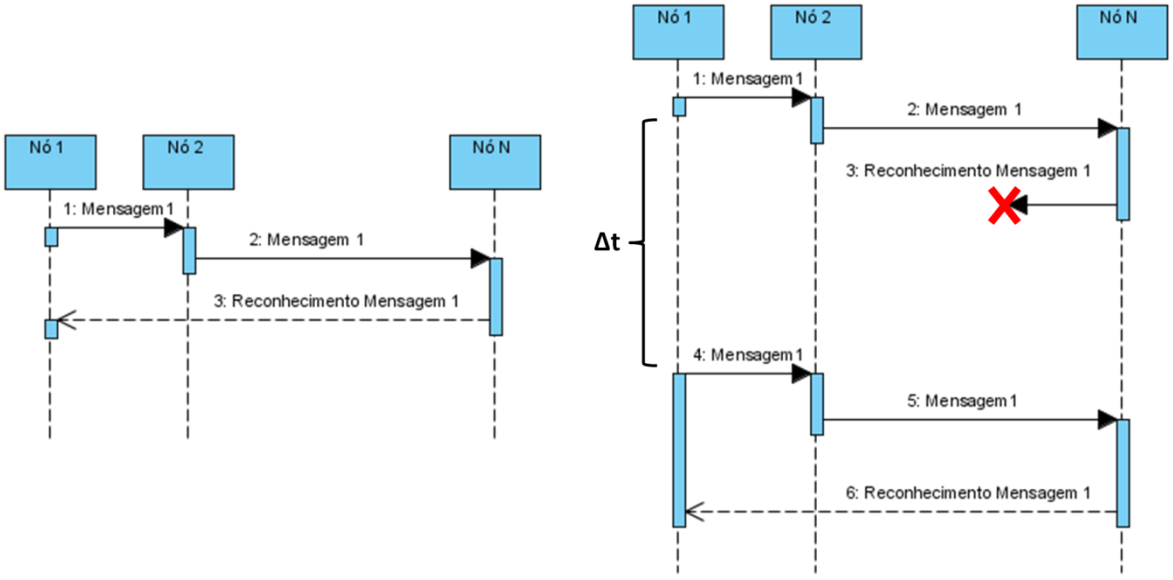


Figura 3.7 - Formato da mensagem

Como, tipicamente, as RdP-IOPT deste trabalho são redes que modelam controladores, é necessário assegurar a fiabilidade, respondendo com uma mensagem de reconhecimento. Apesar de ainda não estar implementado, o protocolo já considera que o transmissor possa vir a ter (dependendo do valor de resposta) que reenviar a mensagem, ou simplesmente ficar prevenido que o destinatário tem o seu *buffer* cheio, e até receber uma mensagem de reconhecimento não lhe enviará mais nada. A Figura 3.8 ilustra dois destes cenários, quando o nó um tem um sinal para enviar ao nó N. O primeiro caso (que já se encontra implementado) demonstra o cenário ideal, em que a mensagem percorre o anel até ao destino e este reenvia uma mensagem de reconhecimento. No segundo caso, a mensagem de reconhecimento é perdida e o nó volta a enviar a mensagem passado um período Δt. Como o código é o mesmo, o nó N não processa o valor do sinal mas volta a enviar a mensagem de reconhecimento. Este código terá uma dimensão de 4 bits, dando assim para 16 mensagens distintas até se repetir. Não é ilustrado na figura, mas se a mensagem do nó 1 se tivesse perdido o

comportamento deste seria o mesmo. No entanto, o receptor em vez de só enviar o reconhecimento, processa a informação do sinal.



a) Cenário normal **b) Cenário com perda de mensagem**

Figura 3.8 - Fluxo de Mensagens

Como a transmissão é feita com base no protocolo lógico já utilizado na norma RS-232, as mensagens a transmitir irão ser compostas por múltiplos de oito bits, mais um de paridade, pelo que terão sempre um tamanho constante múltiplo de 11 bits (considerando *start* e *stop* bits). Este tamanho será determinado pelo número de bits necessários para a codificação dos endereços dos nós envolvidos, e de qualquer dos sinais (dependendo da dimensão do sinal, ficando com o que for maior, e de quantos sinais existem). O tamanho do CRC é definido conforme o número de bits que faltarem para se terem os 11 bits, assim pode dar-se o caso de se ter um CRC de 1 bit, ou de se ter, no máximo, 5 bits. O bit inicial da mensagem a transmitir indicará se a trama a transmitir é, ou não, superior a 8 bits (camada física do modelo OSI), permitindo distinguir entre o primeiro byte e os restantes bytes de uma mensagem (Figura 3.7).

3.5 Nó de rede – Implementação hardware

Na Figura 3.9 é apresentada a estrutura do nó de rede onde se destacam cinco módulos (explicados em detalhe posteriormente). Um para a recepção das tramas série; outro para o envio das tramas série; tem-se a análise da mensagem recebida; e a ligação dos sinais e eventos, quer de entrada quer de saída, com o código de execução associado ao modelo RdP-IOPT (este último obtido directamente através do gerador de código automático). O Comportamento do nó pode ser descrito como:

1. Depois de recebidas as tramas necessárias para se ter qualquer uma das mensagens atrás referidas (Figura 3.7), os bytes recebidos serão retirados do *buffer*;
2. Será analisado se o endereço de destino coincide com o endereço do nó. Se não forem iguais, é sinal que a mensagem não era destinada a este nó mas sim a outro, caso em que a mensagem é colocada no *buffer* de saída (mensagens a enviar);
3. Caso a mensagem seja realmente destinada ao nó de rede, os sinais / eventos e respectivos valores são, depois de retirados da mensagem, colocados nos sinais específicos à entrada de cada modelo. Será também criada uma mensagem de reconhecimento (Figura 3.8) a ser devolvida à origem;
4. Uma vez executado o modelo, as suas saídas serão colocadas num *buffer*. Sempre que este *buffer* não esteja vazio é analisado, uma mensagem é composta e colocada no *buffer* de mensagens a enviar;

Antes de uma análise mais detalhada de cada módulo, descrever-se-á como é implementada a interligação entre os módulos e como foram desenvolvidos os *buffers* utilizados.

Tal como referido, a solução (de cada nó) será composta por vários módulos

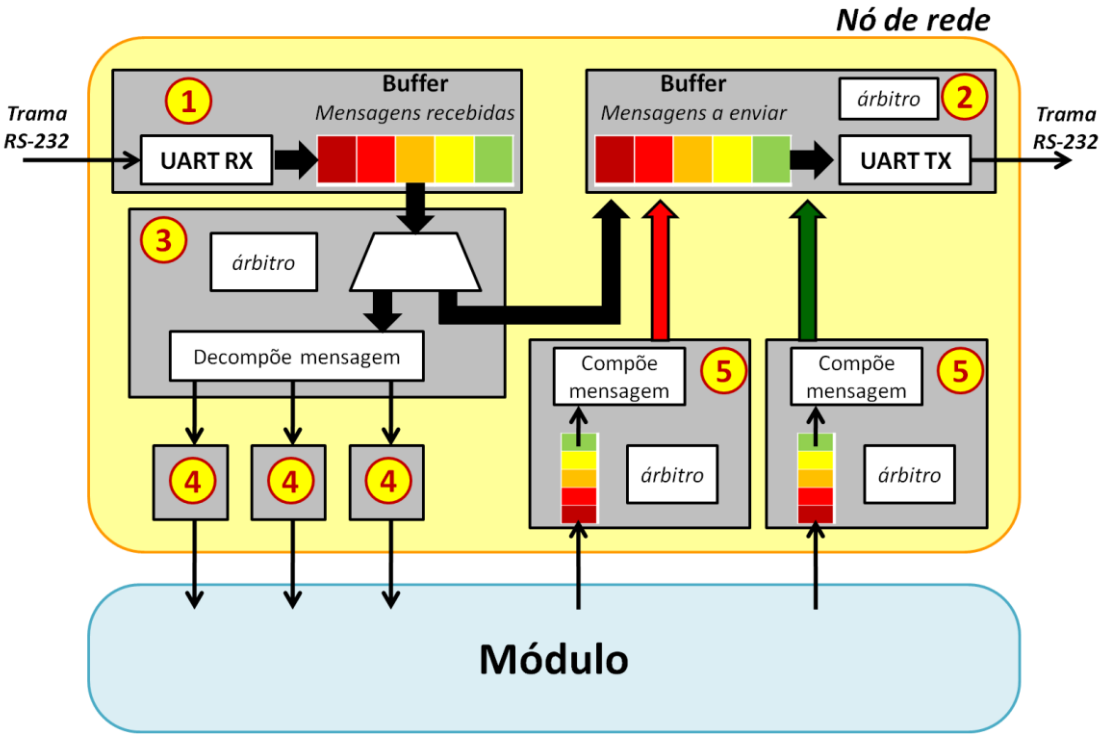


Figura 3.9 - Nó de rede

que comunicam entre si. Nas subsecções que se seguem apresentar-se-á: (1) como estes módulos se interligarão; (2) os *buffers* a utilizar; (3) como serão recebidas as tramas série; (4) como estas serão enviadas; (5) como será analisada a mensagem; (6) como será a interface com os sinais de entrada do módulo; e, finalmente, (7) como será a interface com os sinais de saída do módulo.

3.5.1 Interligação entre blocos

A comunicação entre os (sub)módulos, através de um canal de ligação, é implementada através de um protocolo com *handshake*. A Figura 3.10 ilustra o protocolo entre dois módulos genéricos e as respectivas máquinas de estado utilizadas para a sua implementação. Como se pode verificar pela Figura 3.10, para a sua implementação são necessárias duas componentes. Uma no lado que tem informação para transmitir e outra no que recebe informação. Com o intuito de ao longo deste capítulo se poder fazer referência a estes componentes, a parte da transmissão de dados terá a designação “protocolo handshake A” e a da recepção “protocolo handshake B”.

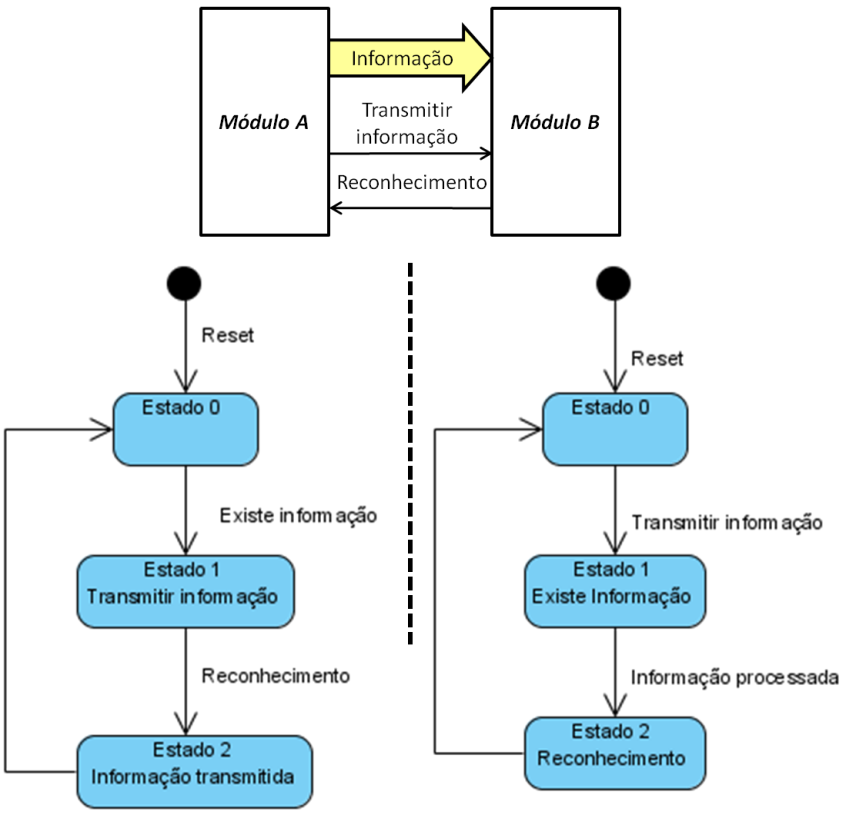


Figura 3.10 - Protolo handshake. Máquinas de estados que implementam o protocolo

A implementação foi feita de um modo síncrono, no entanto o ideal seria ter aplicado máquinas assíncronas, não dependentes do sinal de relógio. A razão pela

qual foi tomada esta opção, na solução proposta neste trabalho, prendeu-se com o facto de o XST (*Xilinx Syntheis Technology*) não fazer a síntese de máquinas de estado finitas assíncronas [38]. Como se pode verificar pela simulação da Figura 3.11, desprezando o tempo de processamento do módulo B, a implementação síncrona terá um custo de dois ciclos de relógio.

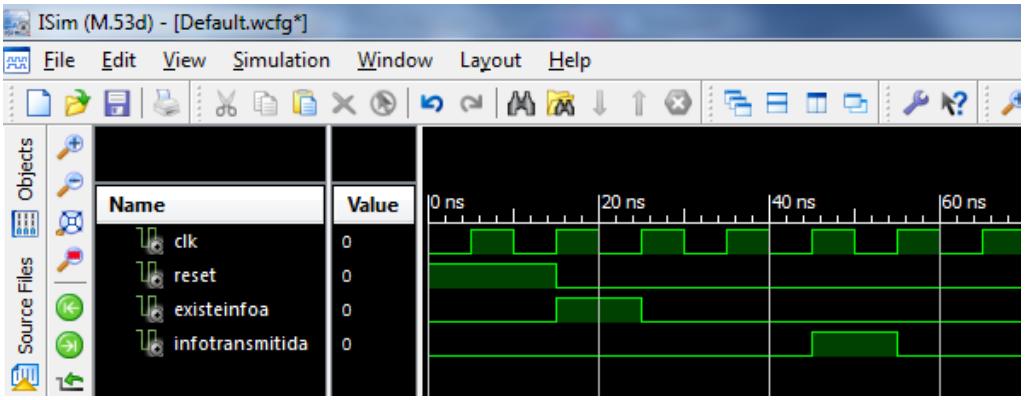


Figura 3.11 - Simulação do protocolo de *handshake* implementado.

Para que o nó de rede seja transparente aos olhos do modelo, não se irá utilizar o protocolo *handshake* na comunicação entre o modelo e os (sub)módulos que interagem directamente com o modelo. Utilizar-se-ão *buffers* onde será acumulada a informação dos sinais / eventos. Os módulos responsáveis por essa interface, num processo concorrente, analisam o *buffer* e depois de processado o sinal comunica com outro módulo usando o protocolo descrito. A Figura 3.12 ilustra esta situação, de forma genérica.

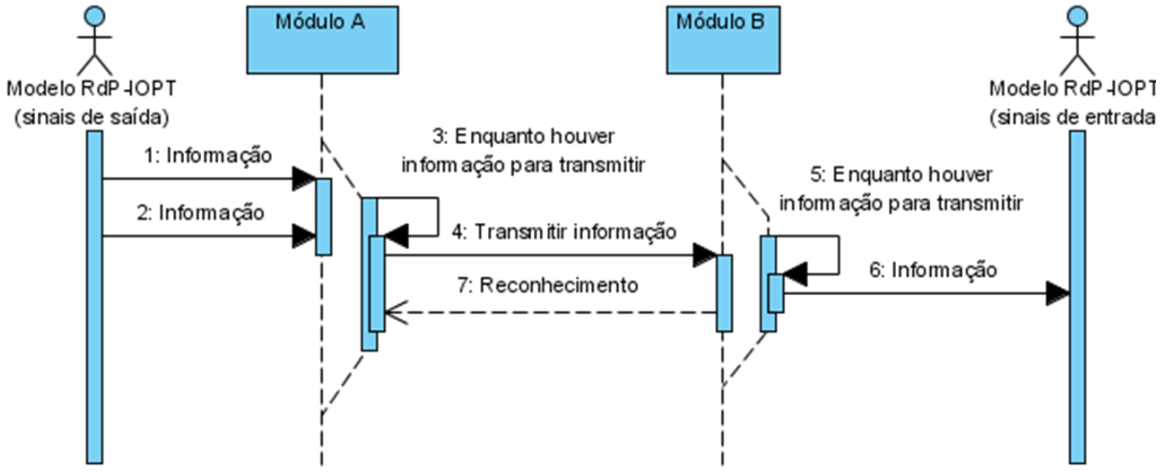


Figura 3.12 - Protocolo de comunicação entre os módulos e os modelos RdP-IOPT

3.5.2 Buffers implementados

Os *buffers* implementados são do tipo FIFO e baseados em registo de deslocamento. A Figura 3.13 ilustra como é feita a sua implementação.

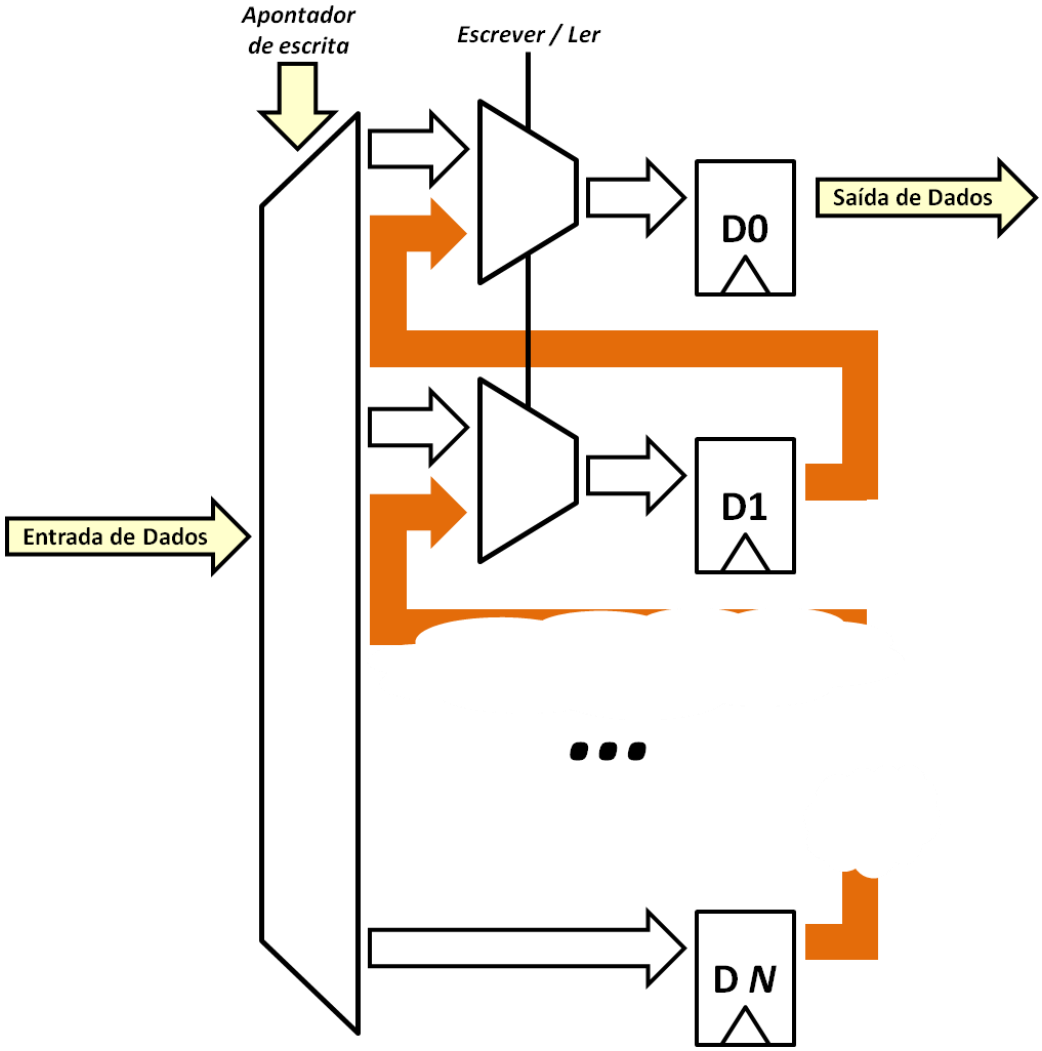


Figura 3.13 - Arquitectura de um FIFO

3.5.3 Recepção das tramas série (bloco 1 da Figura 3.9)

Para a recepção de dados através do protocolo série foi utilizada uma macro UART publicada na nota de aplicação em [39]. A macro UART_RX (Figura 3.14) contém um *buffer* de 16 Bytes, ocupa 8 CLBs (*Configurable Logic Blocks*) e fornece o funcionamento duma UART com as seguintes características:

a) *Start bit*;
b) Oito bits de dados transmitidos em série, começando por receber o bit menos significativo; e
c) *Stop bit*.

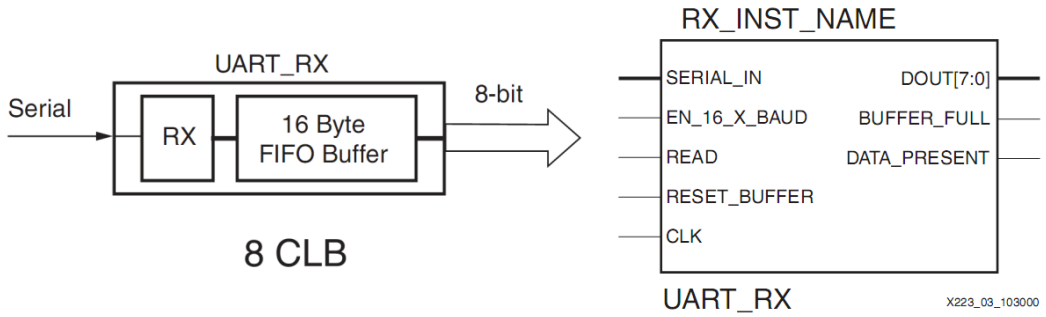


Figura 3.14 - Macro de recepção UART [39]

A Tabela 3.4 descreve de forma resumida o que faz cada sinal, dando a conhecer como esta macro funciona.

Tabela 3.4 - Descrição dos sinais da macro de recepção UART [39]

Sinal	***Direcção***	***Descrição***
CLK	*Entrada*	Sinal de relógio do sistema.
EN_16_X_BAUD	*Entrada*	Esta entrada fornece o *timing* para a transmissão série. Só deve ter o valor lógico UM durante um ciclo de relógio e com uma frequência de 16 vezes (aproximadamente) a velocidade da transmissão de dados série. Se este sinal for definido como UM a velocidade da transmissão série será de CLK/16 bits por segundo.
RESET_BUFFER	*Entrada*	Faz reset ao *buffer* interno de 16 bytes, i.e., todos os dados do *buffer* serão apagados. Se for activo durante uma transmissão série pode corromper os dados recebidos
BUFFER_FULL	*Saída*	Está activo sempre que o *buffer* estiver cheio. A partir desse momento os dados serão perdidos.
DOUT[7:0]	*Saída*	O byte (8 bits) recebido. A informação só é válida quando o sinal DATA_PRESENT está activo.
DATA_PRESENT	*Saída*	Este sinal está activo sempre que o *buffer* interno tenha mais do que um byte recebido
READ	*Entrada*	A entrada activa significa que os dados na saída DOUT[7:0] foram lidos (ou serão lidos no próximo flanco ascendente do sinal CLK) e que os próximos dados existentes podem ser disponibilizados.

Embora o receptor e o emissor sejam assíncronos e não estejam sincronizados, tanto o UART_RX como UART_TX (explicado mais à frente) usam uma referência de tempo com tolerância suficiente que lhes permite a transferência

série dos dados. O receptor usa o flanco descendente do *start bit* para iniciar um circuito interno de gestão de tempo. Esse tempo será usado como amostragem do valor da entrada série no ponto médio (aproximadamente) de cada bit. Esta é a zona onde os dados são mais estáveis (Figura 3.15). Como o receptor se vai sincronizando no flanco descendente de cada *start bit,* o *timing* do transmissor e do receptor deve ser a mesma, com uma precisão de metade de um bit num período de 10 bits (cerca de uma tolerância de 5%).

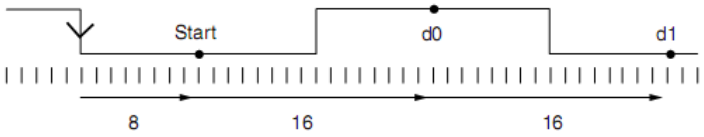


Figura 3.15 - Operação UART – Receptor [39]

Após a recepção do número de bytes necessários para se ter uma mensagem, esta é colocada num *buffer* como o descrito anteriormente. Essa gestão é feita por um controlador implementado através duma máquina de estados como a ilustrada na alínea a) da Figura 3.16. O sinal "DATA_PRESENT" deste *buffer* de mensagens liga ao sinal "Existe informação" do protocolo handshake A, para que posteriormente possa enviar a mensagem ao bloco (3).

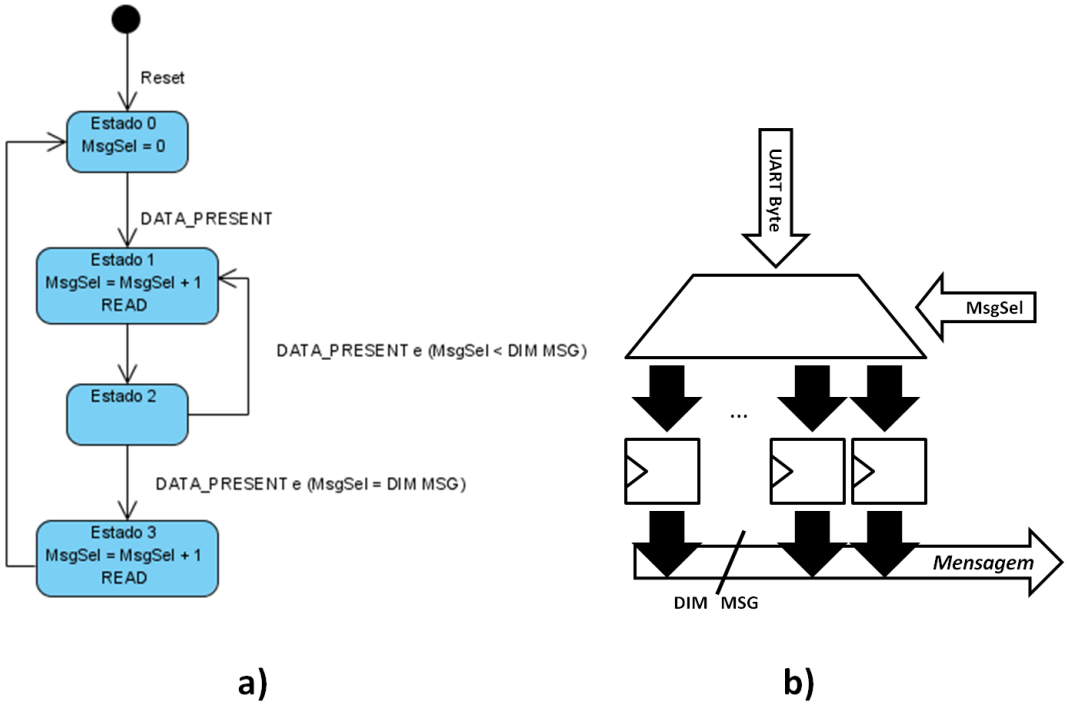


Figura 3.16 - Controlador de mensagens recebidas. a) Parte de controlo. b) Parte de dados

Este módulo terá então como sinais de entrada: (1) o sinal de relógio do nó de rede; (2) sinal de reset; e (3) linha de entrada da comunicação série. Como sinal de saída têm-se as mensagens que estão armazenadas no *buffer*. Para além destes sinais existem mais dois para o protocolo *handshake* com o módulo da análise da mensagem recebida.

Utilizando um *buffer* com uma profundidade para oito mensagens e com um tamanho catorze (equivale a dois bytes recebidos menos os bits que indicam o inicio da mensagem - Figura 3.7) os recursos estimados são os apresentados na Tabela 3.5. Esta tabela, tal como as restantes ao longo deste documento, foi obtida através da aplicação Xilinx ISE Design Suite 12.1, utilizando uma FPGA Spartan-3 XC3S200 FT25, apenas a título indicativo. Uma vez que a tabela foi obtida da ferramenta, está em inglês. Depois da mensagem completamente recebida (após a recepção de todos os bytes que a compõem) demora um ciclo de relógio para colocar os dados no módulo adjacente, e depois de recebido o reconhecimento do protocolo *handshake* leva dois ciclos até voltar ao início do processo.

Tabela 3.5 - Recursos estimados do módulo de recepção de tramas RS-232

Device Utilization Summary	
Logic Utilization	**Used**
Number of Slices	129
Number of Slice Flip Flops	187
Number of 4 input LUTs	211
Number of bonded IOBs	19
Number of GCLKs	1

3.5.4 Envio das tramas série (bloco 2 da Figura 3.9)

Tal como para a recepção de dados através do protocolo série, foi utilizada uma macro UART publicada na nota de aplicação em [39], apresentada na Figura 3.17. A macro UART_TX também contém um *buffer* de 16 Bytes, ocupa 7 CLBs e fornece o funcionamento duma UART com as características descritas para a macro UART_RX.

Os sinais CLK, EN_16_X_BAUD, RESET_BUFFER, BUFFER_FULL são comuns a ambas as macros, e já foram descritos na Tabela 3.4. Os restantes são descritos na Tabela 3.6.

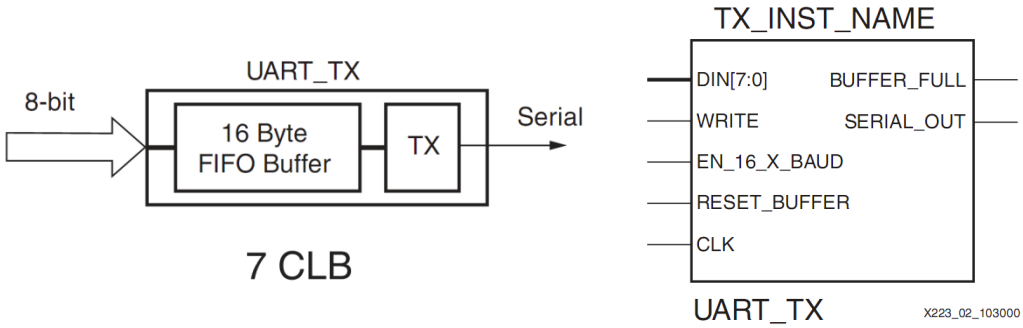


Figura 3.17 - Macro de transmissão UART [39]

Tabela 3.6 - Descrição dos sinais da macro de transmissão UART [39]

Sinal	***Direcção***	***Descrição***
DIN[7:0]	*Entrada*	O byte (8 bits) a enviar na transmissão série. A informação é colocada no *buffer* interno quando o sinal WRITE está activo, durante um ciclo de relógio.
WRITE	*Entrada*	Este sinal será activo sempre que se quiser colocar informação, no próximo flanco ascendente do sinal de relógio, no *buffer* interno para que seja enviada a informação.
SERIAL_OUT	*Saída*	São os dados série, como referido atrás, um *start bit*, oito bits de dados (contidos no *buffer*) e um *stop bit*. Até o *buffer* estar vazio os dados serão transmitidos.

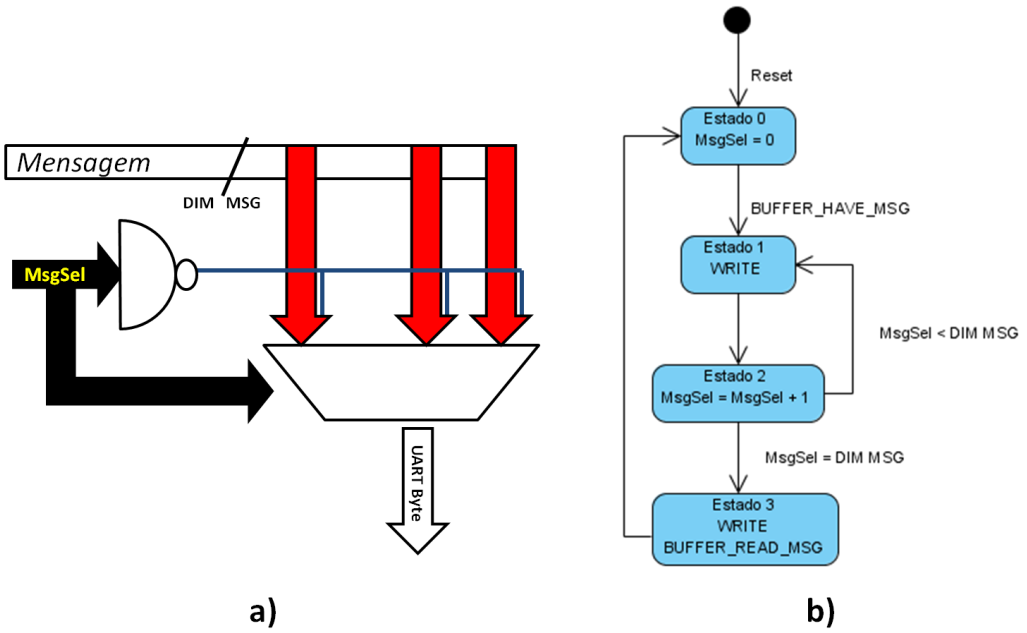


Figura 3.18 - Controlador de mensagens a enviar. a) Parte de dados. b) Parte de controlo.

O processo de colocar uma mensagem no *buffer* é análogo ao descrito na recepção de tramas série, mas de forma inversa, ou seja, mal se tenha uma

mensagem no *buffer* esta é decomposta em N bytes, sendo N o número de bytes necessário para se ter uma mensagem (Figura 3.7), e estes colocados na macro UART_TX descrita. Este mecanismo é ilustrado na Figura 3.18. A porta NAND utilizada serve para acrescentar o bit necessário para indicar se se está perante o primeiro byte da mensagem.

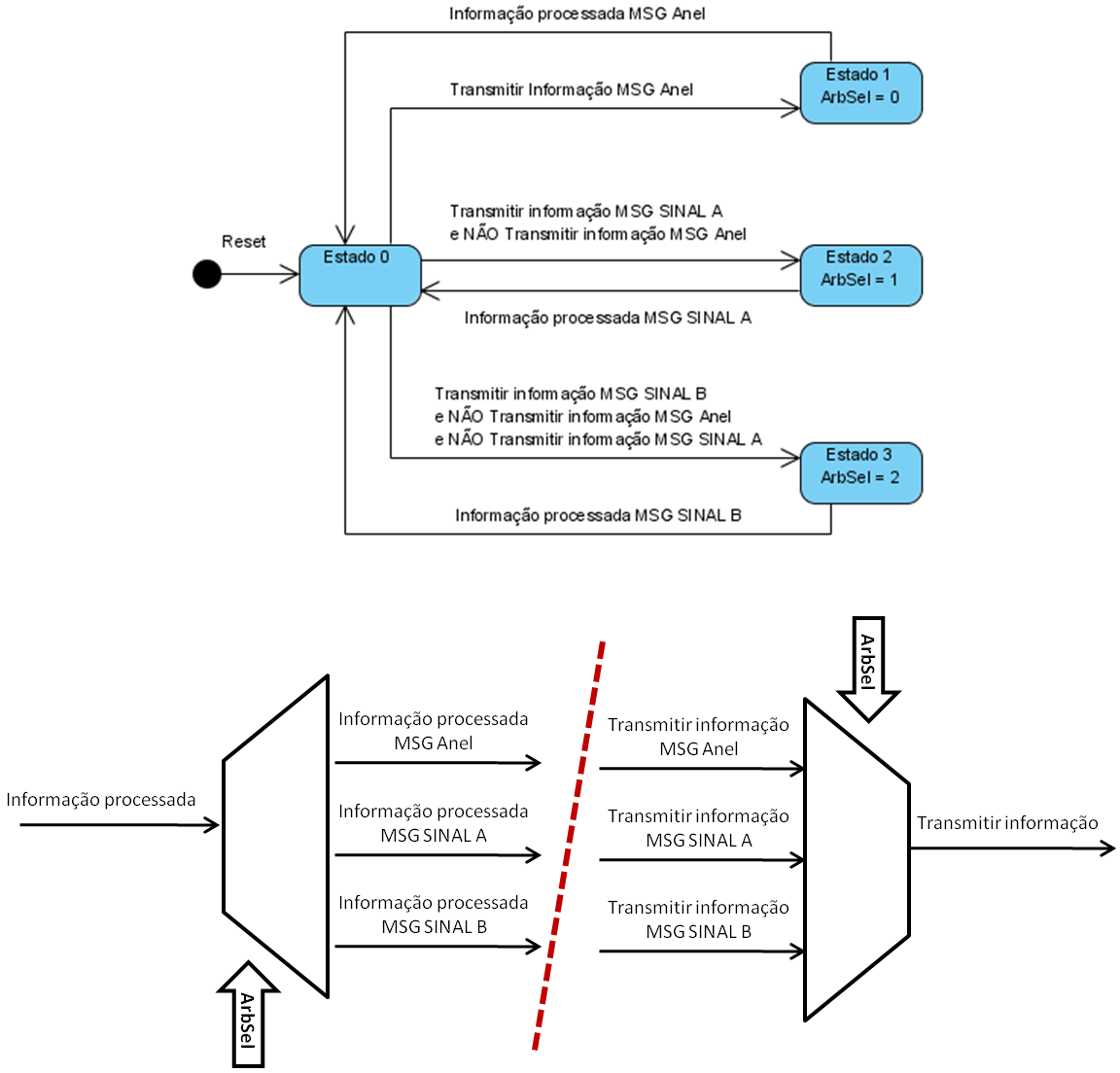


Figura 3.19 - Árbitro de selecção das mensagens a enviar

Para colocar a mensagem neste mecanismo descrito, é necessário colocá-la previamente no *buffer* FIFO para o efeito. É de notar que, este módulo pode receber mais do que uma mensagem. Uma delas vinda do anel, e um número incerto, dependendo do sistema modelado, proveniente dos módulos inicialmente expressos em RdP. Todos esses módulos comunicam com este usando o protocolo *handshake* e é necessário um árbitro que atribuirá a vez (uma espécie de testemunho - *token*) a cada uma das mensagens para as colocar no *buffer*. Ou seja, ter-se-á o mesmo número de componentes que implementam o protocolo handshake B que o mesmo

número de mensagens. O árbitro selecciona qual deles deverá “processar” a informação. Na Figura 3.19 é mostrado como é implementado o árbitro proposto quando se têm dois sinais de saída no módulo.

No árbitro, ao contrário do que foi explicado até então, não bastam variáveis genéricas (no VHDL) para que seja dinâmico a qualquer sistema. A máquina de estado, assim como os *multiplexers*, selecciona qual o módulo que transmite a mensagem e estabelece o protocolo de *handshake* com o *buffer* de mensagem que “fornece” a UART.

Assim, a regra para se construir a máquina de estados do árbitro será:

1. Saber quais (e quantos são) os sinais que vêm do modelo;
2. O total de estados a considerar é o número de sinais mais dois (estado inicial e estado para a mensagem recebida da rede);
3. À excepção do primeiro estado, cada estado está associado a um módulo, donde vem a mensagem, que tem como saída um valor para ArbSel.
4. A condição do primeiro estado (inicial) para o estado de cada módulo é ter recebido “Transmitir informação” do protocolo *handshake* do respectivo módulo;
5. A condição de se sair de cada um desses estados para o estado inicial é receber o respectivo sinal “Informação processada”.

Este módulo terá como sinais de entrada: (1) o sinal de relógio do nó de rede; (2) sinal de reset; e (3) as mensagens vindas dos módulos 3 e 5 representados na Figura 3.9. Como sinal de saída existe a linha da comunicação série. Para além destes sinais, existem os outros sinais para o protocolo *handshake* com os módulos referidos.

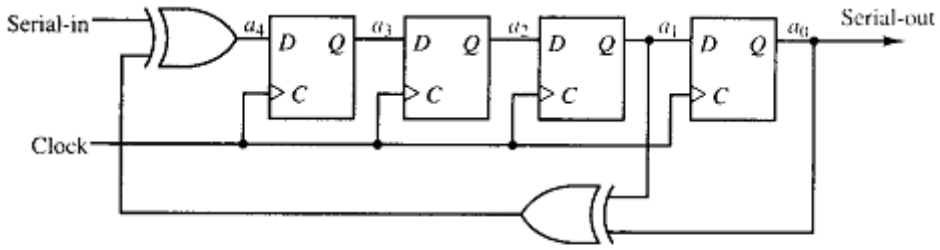
Continuando a utilizar um *buffer* com uma profundidade para oito mensagens e com um tamanho de catorze (equivale a dois bytes enviados menos os bits que indicam o inicio da mensagem - Figura 3.7), e com dois sinais, o que resulta em duas possíveis mensagens, os recursos estimados são os apresentados na Tabela 3.7. Este módulo leva somente dois ciclos de relógio por cada mensagem que tem que enviar. No caso de estar presente mais do que uma mensagem ao mesmo tempo, leva um ciclo entre uma e outra. Ou seja, se se tiver duas mensagens, o módulo leva três ciclos a colocá-las na UART e não quatro.

Tabela 3.7 - Recursos estimados do módulo de envio de tramas RS-232

Device Utilization Summary	
Logic Utilization	**Used**
Number of Slices	137
Number of Slice Flip Flops	168
Number of 4 input LUTs	254
Number of bonded IOBs	51
Number of GCLKs	1

3.5.5 Análise da mensagem recebida (bloco 3 da Figura 3.9)

Antes da análise da mensagem esta é decomposta nos sinais nela contidos (Figura 3.7). Depois desta etapa analisa-se o CRC para se saber se a mensagem está corrompida ou se é válida. Existem várias propostas de implementação para este tipo de verificações. A adoptada foi a *serial input signature register* (SISR) retirada de [5], ilustrada na Figura 3.30, para um CRC de quatro bits. É uma proposta elegante e de fácil implementação. O número de bits do CRC será o número de *flip-flops*, e os dados serão colocados de forma série (bit a bit) pois o mecanismo deste sistema baseia-se num registo de deslocamento. Isto implica que para a análise do CRC seja necessário o mesmo número de ciclos de relógio que o



(a)

Serial in $z(x)$	LFSR Contents a_3	a_2	a_1	a_0
0	0	0	0	0
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
1	1	0	0	1
1	0	1	0	0
0	1	0	1	0
1	1	1	0	1
1	0	1	1	0
1	0	0	1	1
0	1	0	0	1
$r(x)$ =	1	1	0	0

(b)

Serial in $z(x)$	LFSR Contents a_3	a_2	a_1	a_0
0	0	0	0	0
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
1*	0	0	1	0
1	0	0	0	1
1	0	0	0	0
0	1	0	0	0
1	0	1	0	0
1	1	0	1	0
1	0	1	0	1
0	0	0	1	0
$r^*(x)$ =	1	0	0	1

(c)

Figura 3.20 - Operação SISR [5]. (a) diagrama (b) resposta a uma sequência correcta (c) resposta a uma sequência corrompida

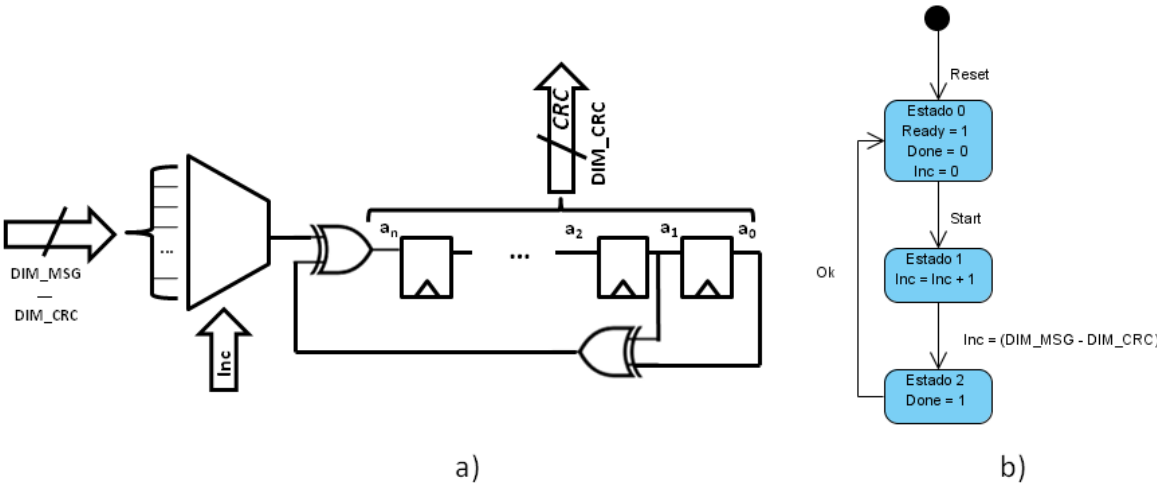


Figura 3.21 - Módulo de verificação do CRC implementado. a) SISR; e b) controlador da verificação CRC

número de bits que compõem a mensagem menos o número de bits do CRC. Uma vez obtido o CRC compara-se com o extraído da mensagem; se forem iguais é analisada a mensagem, caso contrário será descartada. A Figura 3.21 mostra, para o caso de N bits, como foi implementada a análise do CRC. A alínea a) ilustra o SISR e a b) o seu controlador. Este bloco tem duas variáveis (*Ready* e *Done*) que permitem saber se é possível realizar a operação e se esta já foi concluída. Nesse caso é porque já foram dados todos os ciclos de relógio necessários e já se pode considerar o CRC. A análise do CRC começa quando é activo o sinal *Start*.

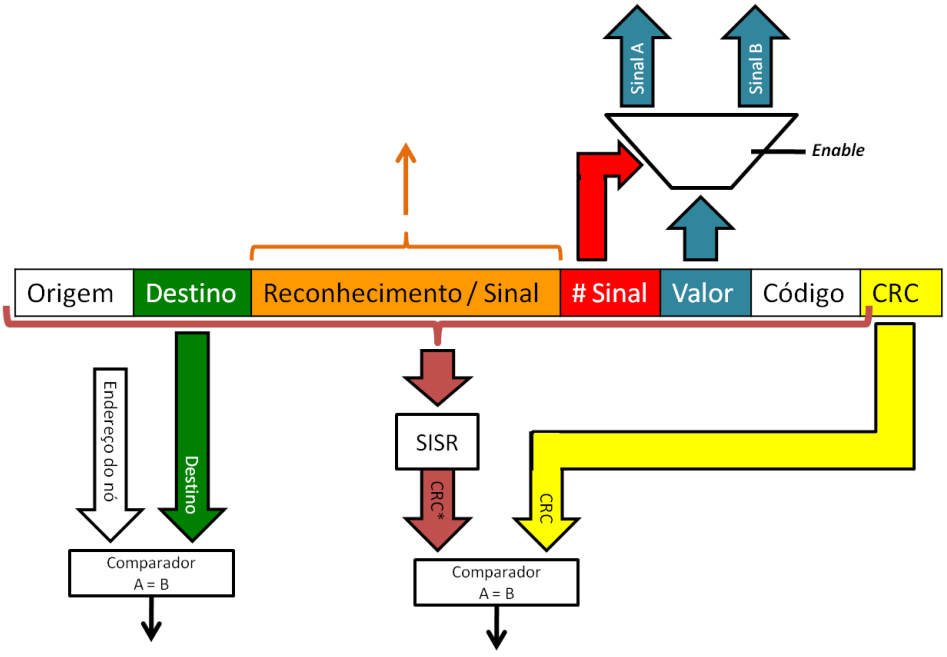


Figura 3.22 - Arquitectura da análise da mensagem

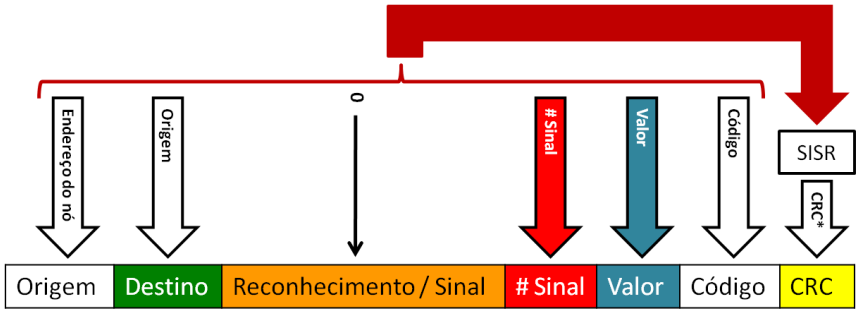


Figura 3.23 - Composição da mensagem de reconhecimento

O resto do funcionamento deste módulo é ainda mais simples. Primeiro é verificado através de um comparador se o sinal destino corresponde ao seu endereço (guardado no sinal genérico no VHDL). Se não, coloca no *buffer* do módulo (colocando o sinal "Existe informação" do protocolo handshake A activo, ficando o bloco (3) da Figura 3.9 "parado" até que seja recebido o reconhecimento do mesmo protocolo) de envio de tramas série, descrito atrás; se sim (Figura 3.22), através do sinal enable, é a componente com a codificação do sinal que controla tanto os *multiplexers* que gerem os dados, como o protocolo *handshake* A que comunica com os respectivos módulos (módulo 4 da Figura 3.9). Para além de enviar os sinais, é também enviada uma mensagem de reconhecimento (mais uma vez através do protocolo handshake A) composta com base nos campos recebidos pela mensagem original. A excepção será a origem, que será o endereço do nó, o campo de reconhecimento que passará para zero, e será calculado um novo CRC (Figura 3.23). O controlador apresentado na Figura 3.24 mostra como foi implementada esta gestão.

Este módulo terá como sinais de entrada: (1) o sinal de relógio do nó de rede; (2) o sinal de reset; e (3) as mensagens vindas do módulo 1 representado na Figura 3.9. Como sinal de saída existe o sinal extraído da mensagem. Para além destes sinais, existem os outros sinais para o protocolo *handshake* com os módulos referidos.

Utilizando uma mensagem com catorze bits, e com dois sinais de entrada no modelo, os recursos estimados são os apresentados na Tabela 3.8. O tempo de processamento deste módulo varia. Para além de dois ciclos de relógio para os protocolos *handshake* com os módulos adjacentes, precisa do mesmo número de ciclos de relógio que o número de bits da mensagem menos o tamanho do CRC para o validar. Como o modelo pode ser implementado com um ritmo de relógio diferente e desfasado, demora mais esse tempo.

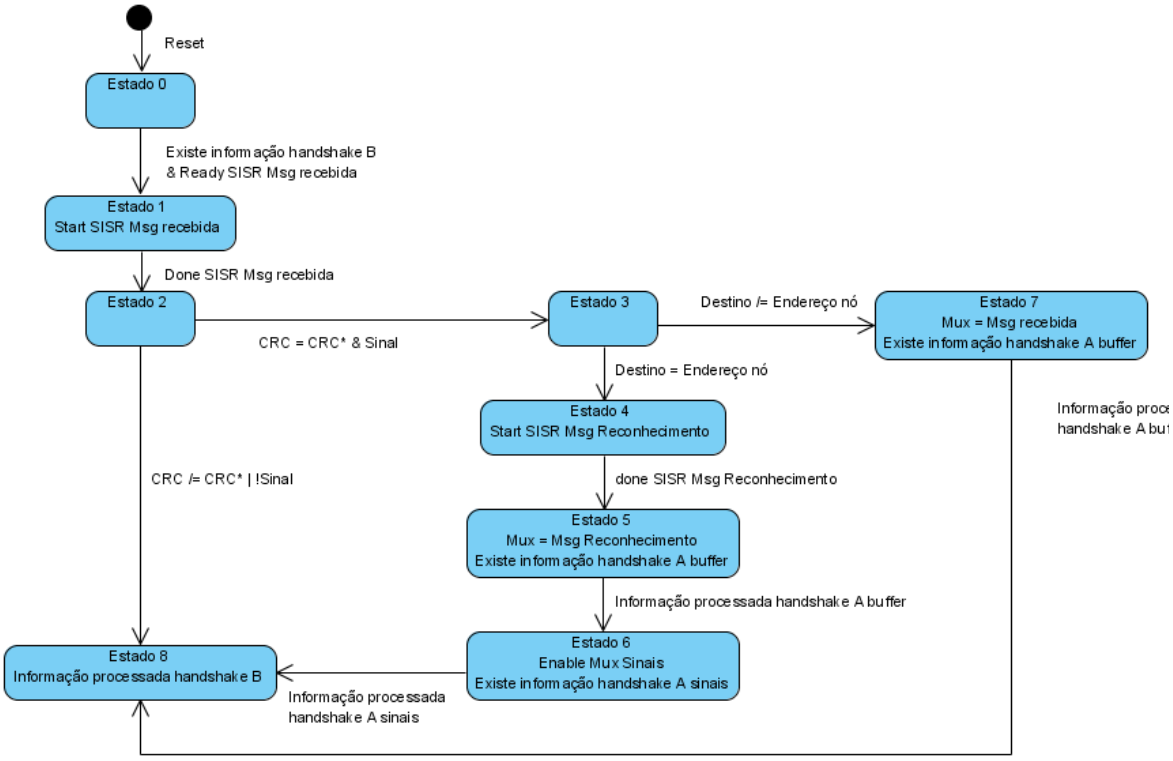


Figura 3.24 - Controlador da analise da mensagem recebida

Tabela 3.8 - Recursos estimados do módulo de análise da mensagem recebida

Device Utilization Summary	
Logic Utilization	**Used**
Number of Slices	14
Number of Slice Flip Flops	4
Number of 4 input LUTs	23
Number of bonded IOBs	40
Number of GCLKs	1

3.5.6 Interface com os sinais / eventos de entrada do módulo (bloco 4 da Figura 3.9)

Este módulo consiste unicamente em aplicar o protocolo *handshake* mas com uma particularidade, a solução proposta permite que o nó de rede funcione com um sinal de relógio diferente do módulo inicialmente modelado com RdP. Isto porque o sistema a implementar pode exigir velocidades diferentes entre os vários submodelos, e o nó deve ser o mais rápido possível para que a transmissão de mensagens custe o menor atraso possível. Assim, e para que o nó seja "transparente", no modelo teve que se ter uma máquina que funcionasse com a velocidade de relógio do modelo e que implementasse o modelo *handshake* atrás descrito. Apesar dos sincronizadores apresentados em [27] serem mais complexos,

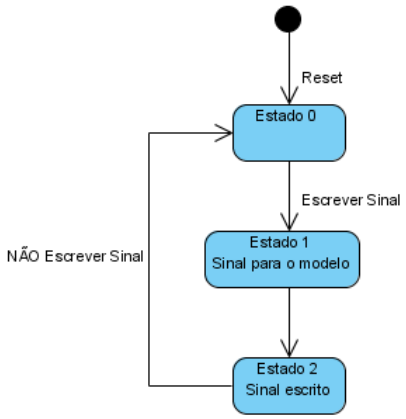


Figura 3.25 - Sincronizador da interface com os sinais / eventos do módulo

pode-se ver, abusando do termo, a máquina de estados apresentada na Figura 3.25 como o sincronizador entre a informação vinda do nó e a velocidade de execução do modelo.

Este controlador, que funciona com o sinal de relógio do modelo em vez do do nó, sempre que o sinal "*Transmitir informação"* vindo do outro módulo (bloco 3 da Figura 3.9) estiver activo, activará o sinal "Escrever Sinal". No flanco ascendente do relógio do modelo será enviado o sinal / evento ao módulo (através do sinal "Sinal para o modelo" que controla um *buffer* tristate) e terá a duração de um ciclo de relógio. A Figura 3.26 mostra o resultado da solução proposta, com sinais de relógio diferentes e desfasados. O sinal de relógio do sistema tem um período de 10 ps e o sinal de relógio do modelo tem um período de 22 ps.

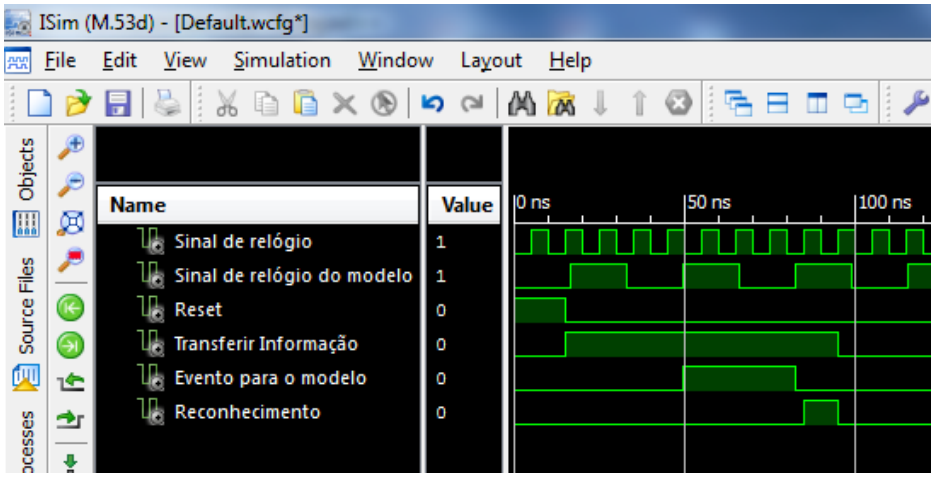


Figura 3.26 - Simulação do sincronizador da interface com os sinais

Este bloco só tem como sinais de entrada e de saída, para além dos sinais de *handshake* do reset e do sinal de relógio do sistema e do modelo, o sinal a enviar ao módulo com um custo associado apresentado na Tabela 3.9.

Tabela 3.9 - Recursos estimados do módulo de interface com os sinais / eventos de entrada do módulo

Device Utilization Summary	
Logic Utilization	**Used**
Number of Slices	2
Number of Slice Flip Flops	4
Number of 4 input LUTs	4
Number of bonded IOBs	6
Number of GCLKs	2

3.5.7 Interface com os sinais / eventos de saída do módulo (bloco 5 da Figura 3.9)

Este módulo é composto por duas zonas. Uma, responsável pela interface com o modelo, e outra, encarregue de transformar o sinal numa mensagem. A primeira tem que ter as considerações descritas há pouco, de modo a ser possível captar a informação de forma transparente ao modelo, mesmo que ambos tenham eventualmente velocidades de relógio diferentes. Neste caso, também não se pode ter um protocolo *handshake,* pois não seria transparente ao modelo e poder-se-ia perder informação. Assim, teve que se utilizar uma solução já explorada em sistemas GALS (*Globally Asynchronous, Locally Synchronous*). As GALS têm diversas interfaces [40], sendo a melhor opção, a que mais se adequa à resolução deste desafio, uma interface assíncrona. Este método usa circuitos sincronizadores para transferir os sinais de um domínio do tempo para outro.

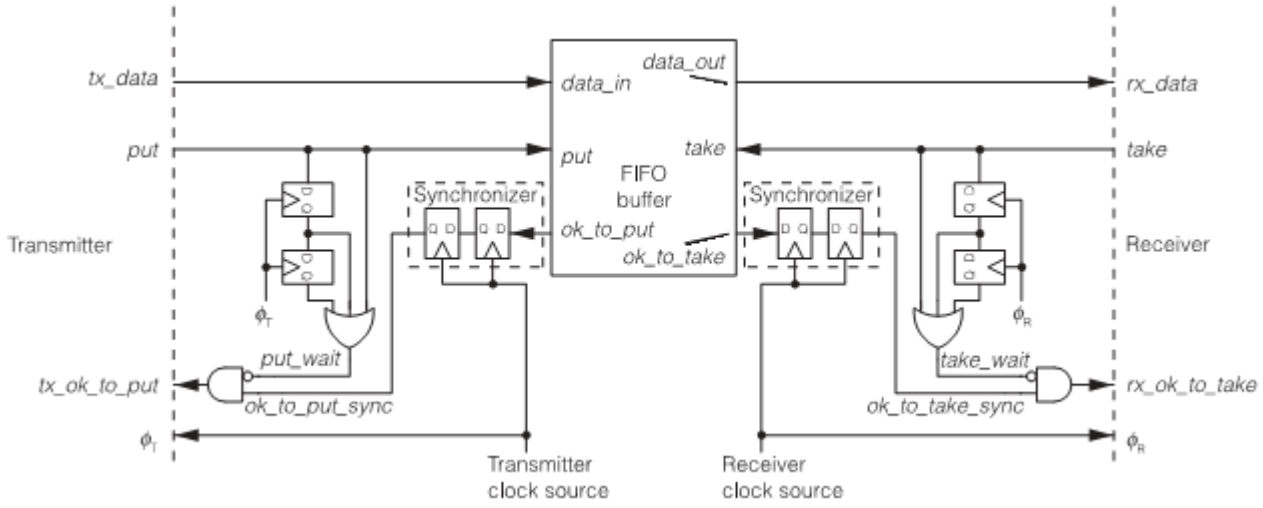


Figura 3.27 - Design dum sistema GALS de estilo assíncrono, usando sincronizadores [40].

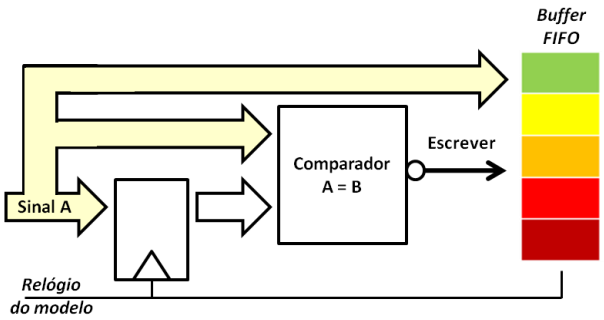


Figura 3.28 - Variação do valor do sinal e respectiva inserção no buffer

A Figura 3.27 mostra o design de um sistema GALS com uma interface assíncrona. Apesar de o esquema não ser igual ao implementado, representa a base do que foi desenvolvido. Na perspectiva deste trabalho, e como não existe um protocolo *handshake* com o modelo, não é necessário um dos sincronizadores. O módulo tem um *buffer* FIFO, apresentado em 3.5.2. O sinal de relógio do *buffer* é o mesmo do modelo RdP. Sempre que existir um evento ou ocorrer uma variação no sinal é colocado no *buffer*. A Figura 3.28 ilustra o esquema de como foi implementada a detecção da variação de sinal, e respectiva inserção no *buffer* (é a adaptação realizada no sistema da Figura 3.27, à esquerda do *buffer*). Caso se trate de um evento, este é directamente ligado ao *buffer* porque existe o interesse de guardar todos os eventos ocorridos. Com este método, deixa de ser necessário a lógica e o sincronizador do lado esquerdo do esquema apresentado. Sempre que existir informação no *buffer* tem que ser retirada com uma velocidade igual à do modelo (velocidade de relógio do *buffer*). Assim existe um sincronizador (ilustrado na Figura 3.29) entre o protocolo *handshake*, depois de confirmar que as mensagens compostas já foram para o módulo responsável pelo envio das tramas série (utilizado com o relógio do sistema), e o *buffer*. Este sincronizador é pouco ortodoxo dado que, explorando a ligação com o protocolo *handshake*, consiste numa máquina

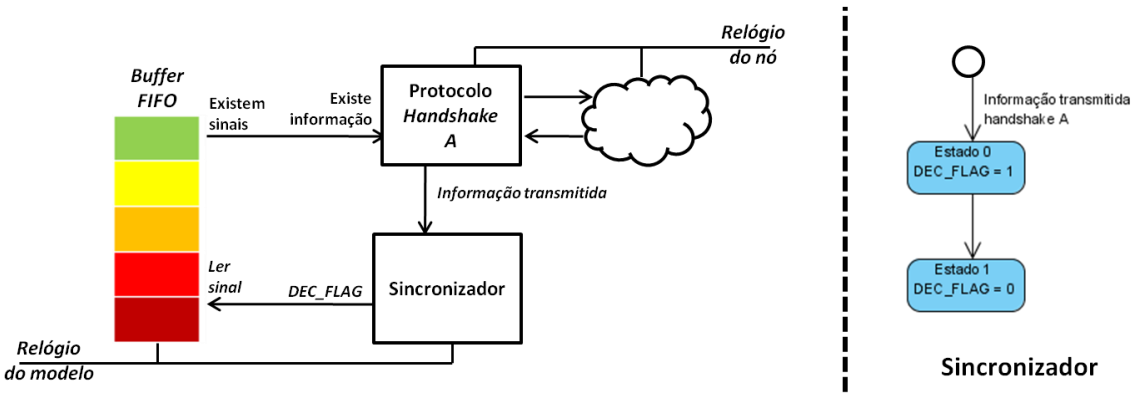


Figura 3.29 - Sincronizador para retirar elemento do buffer

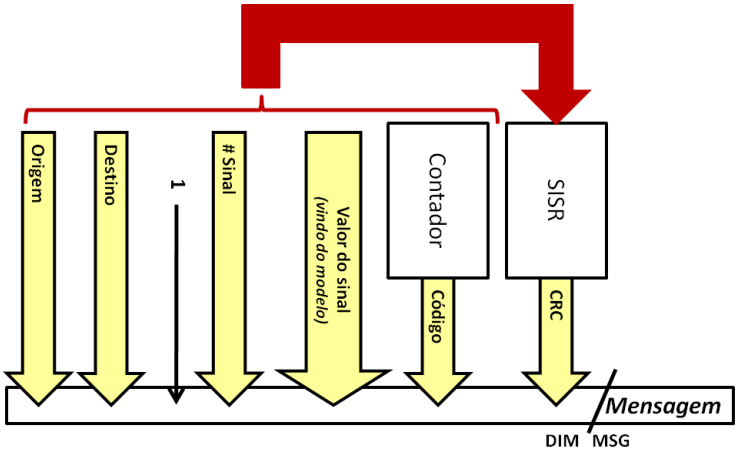


Figura 3.30 - Diagrama de como é feita a composição de uma mensagem a partir de um sinal vinda do modelo.

de estado síncrona com a velocidade de relógio do modelo que fica "preso" num estado até que o sinal "informação transmitida" do protocolo *handshake* A tenha o comportamento de reset assíncrono (é a adaptação realizada no sistema da Figura 3.27, à direita do *buffer*).

A transformação do sinal em mensagens é um processo simples. À partida, sabe-se que o sinal terá que ser entregue em um ou mais módulos. Dessa forma, sabe-se quantas mensagens terão que ser construídas e quais as codificações dos respectivos nós de rede e codificações do sinal. Este módulo tem sinais genéricos onde está essa informação.

As mensagens formadas terão o formato apresentado atrás (Figura 3.7). Existe um módulo contador que a cada mensagem enviada é incrementado (através do sinal "Informação processada"), que será utilizado para se criar o código e só depois é construído o CRC pelo mesmo método explicado, para que finalmente se tenha cada mensagem final. A Figura 3.30, semelhante à já exposta anteriormente em Figura 3.23, ilustra como este processo é feito.

Depois de se terem todas as mensagens a enviar, há que ter em atenção que só se pode voltar a analisar a existência de novos sinais quando todas elas forem colocadas no módulo de envio de tramas RS-232. Assim, existe um contador assíncrono que é incrementado cada vez que existe "um *handshake*" com esse módulo, e a informação só será dada como processada quando o contador for igual ao número de mensagem a enviar. O contador também serve para seleccionar a mensagem a enviar. A Figura 3.31 mostra as máquinas de estado que controlam o procedimento descrito.

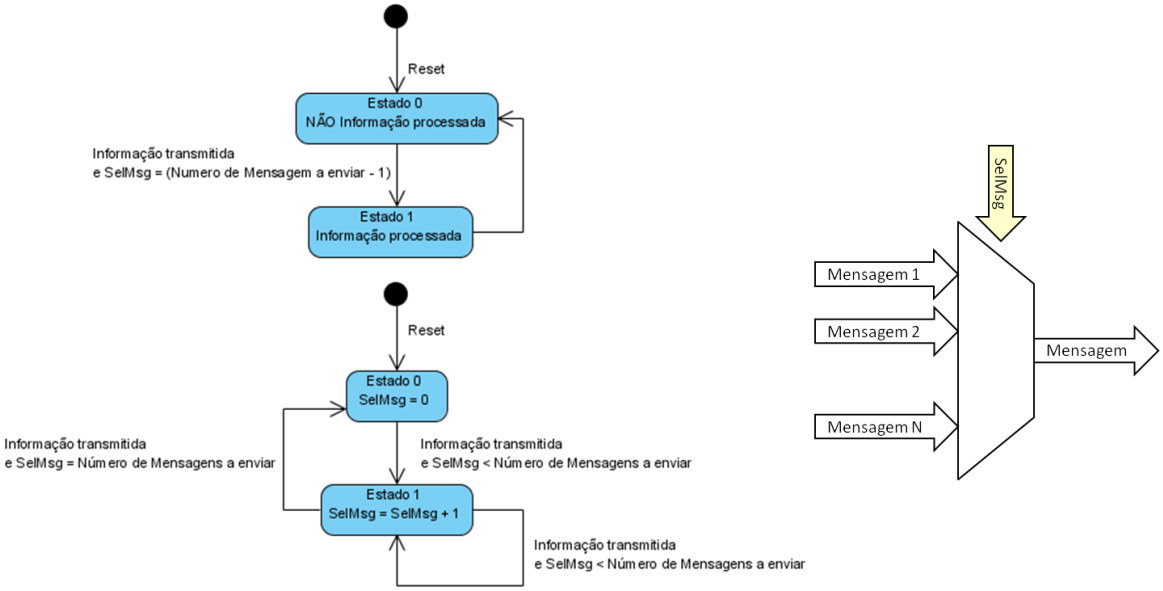


Figura 3.31 - Controlador do número de mensagens enviadas

Este módulo terá como sinais de entrada: (1) o sinal de relógio do nó de rede; (2) o sinal de relógio do modelo; (3) o sinal de reset; e (4) o sinal / evento vindo do modelo. Como sinal de saída, existe a mensagem que irá para o módulo encarregue de enviar as tramas série, para além dos sinais para o protocolo *handshake*.

Utilizando mensagens de catorze bits, e com o sinal a ter que chegar a três modelos (o que implica a composição de três mensagens), os recursos estimados são os apresentados na Tabela 3.10.

Tabela 3.10 - Recursos estimados do módulo de interface com os sinais / eventos de saída do módulo

Device Utilization Summary	
Logic Utilization	**Used**
Number of Slices	15
Number of Slice Flip Flops	20
Number of 4 input LUTs	28
Number of bonded IOBs	20
Number of GCLKs	2

3.6 Ponte – Implementação hardware

A Figura 3.32 descreve a estrutura da ponte, onde se pode verificar que os módulos referidos são os mesmos que os apresentados na secção anterior. Esta estrutura não comunica directamente com o modelo desenvolvido e sim com a plataforma, ou seja, comunica com um nó de rede como o descrito anteriormente.

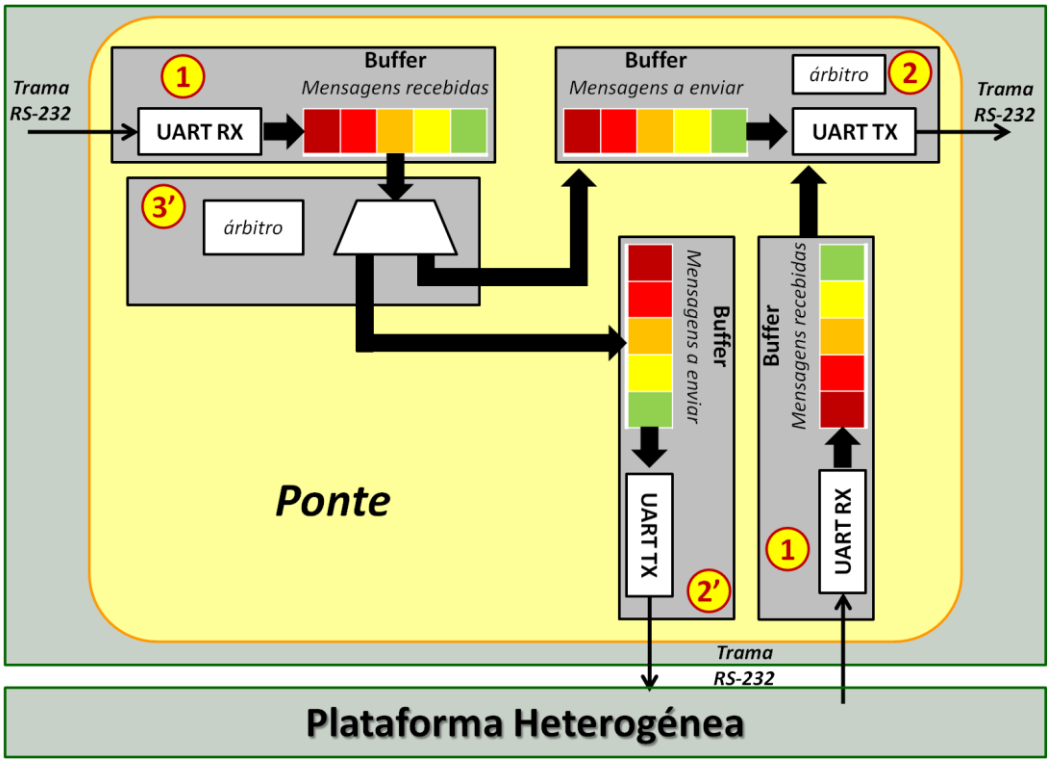


Figura 3.32 - Ponte entre plataformas heterogéneas

O funcionamento é semelhante ao nó de rede. Os blocos (1) e (2) já foram explicados em 3.5.3 e 3.5.4, respectivamente. Os blocos (4) e (5) da Figura 3.9 desaparecem, dado serem os blocos responsáveis pela interface com o modelo, o que neste caso não acontecerá. Em vez disso, tem-se o bloco (1), que recebe tramas série vindas de um nó e as transforma em mensagens e o bloco (2'). O bloco (2') é igual ao bloco (2) mas não é necessário o árbitro (exemplificado na Figura 3.19), basta o protocolo *handshake*, pois só recebe uma mensagem, convertendo-a no final em tramas série. O bloco (3) também foi alterado, ficando com a designação (3'). Como já não é necessária a descodificação completa da mensagem para posterior envio dos sinais / eventos, só é analisado o CRC e, caso este esteja correcto, verifica se o endereço de destino corresponde ao endereço da ponte (codificado da mesma forma que o nó de rede). Consoante esse resultadoa mensagem é colocada no bloco (2), ou no (2').

3.7 Nó de rede – Implementação Software

Quando se trata de plataformas *hardware,* a ligação da ponte pode ser feita tanto com outra ponte como com um nó de rede (se só existir um modelo na segunda plataforma). O esquema da Figura 3.33 mostra onde se situará o nó de rede caso a implementação seja feita em *software*. É de realçar que, como o nó tem que fazer a interface com o modelo, previamente convertido de RdP-IOPT para C de

forma automática, e usa UARTs, está-se dependente do compilador. Isto é, apesar do C obtido pelo gerador PNML2C ser ANSI C para facilitar a compatibilização com os compiladores, o mesmo não acontece na interface com as UARTs, a qual está dependente da plataforma e do compilador utilizado.

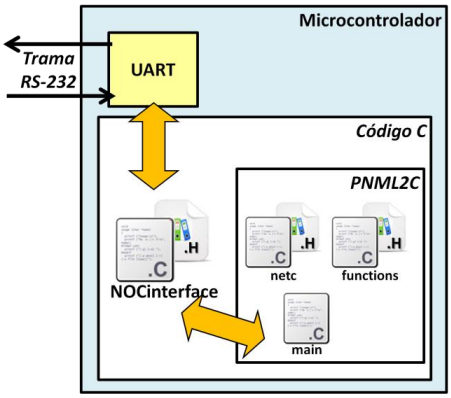


Figura 3.33 - Nó de rede com implementação em software.

O nó de rede consiste no conjunto de ficheiros (.C e .H) *NOCinterface* e fornece uma série de funções que implementam o protocolo referido. Cada sinal do módulo tem associado um *array*, com um apontador de escrita, que fará de *buffer*. A leitura é sempre realizada a partir do índice zero, e mal seja feita todos os dados são deslocados uma posição.

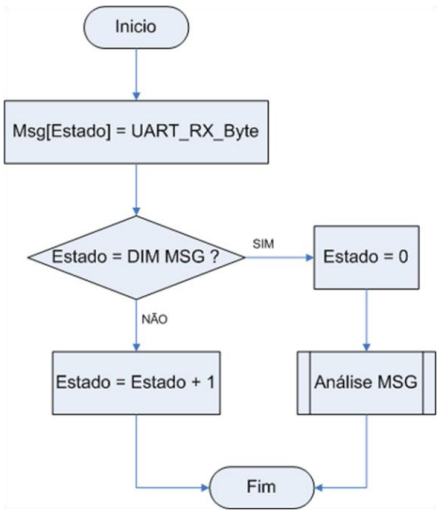


Figura 3.34 - Fluxograma da função associada ao interrupt da UART

Para que a interface desenvolvida não prejudique o desempenho da execução do código gerado pelo PNML2C, foram utilizados os *interrupts* da UART sempre que se recebe um byte. Tal como ilustrado no fluxograma da Figura 3.34, existe uma variável MSG do tipo *array* de bytes que vai recebendo os dados da UART, e mal se tenha a mensagem completa, ela é analisada.

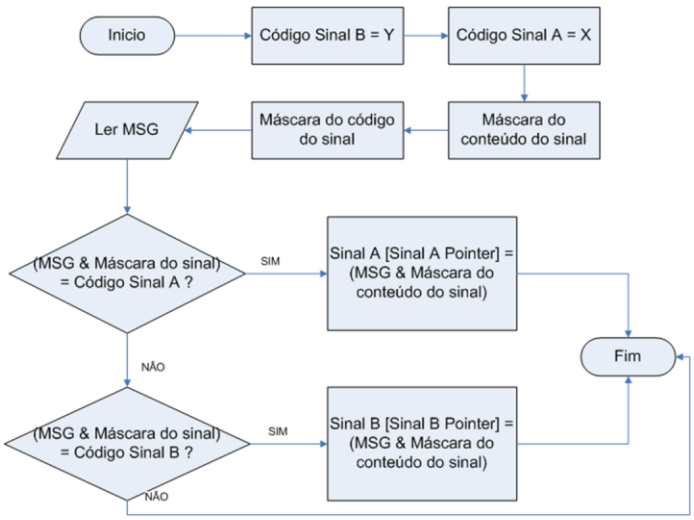


Figura 3.35 - Fluxograma da análise da mensagem recebida ao nó.

Para a análise da mensagem, têm-se variáveis com o código de cada sinal e máscaras para extrair (através de E lógicos) o que se quer analisar. Será depois colocado o valor do sinal no *buffer* respectivo, Figura 3.35.

Existe mais uma função que consiste em colocar os sinais / eventos do modelo numa mensagem e, depois de decompô-la, enviar byte a byte para a UART. Assim, onde antigamente se liam os sinais / eventos, agora chama-se a função que retirará do *buffer* respectivo o último valor recebido e, onde se escreviam os sinais agora chama-se a função que compõe a mensagem.

4 Exemplos de aplicação

Este capítulo demonstra como aplicar a solução proposta, de forma manual, em dois exemplos de aplicação. Primeiro serão descritas as plataformas onde se fará a implementação e posteriormente identificar-se-á, para cada exemplo, a solução descrita e os resultados obtidos.

4.1 Plataformas e ambiente de experimentação

Antes de se prosseguir com os exemplos de aplicação, esta secção abre um parêntesis para explicar um pouco as plataformas que serão utilizadas para a aplicação dos mesmos.

Um dos objectivos deste trabalho é implementar a interface descrita em duas plataformas, quer em hardware (através de VHDL), quer em software (código C). As plataformas escolhidas foram a Spartan-3 Starter Kit Board, da Xilinx, e o kit didáctico XUPV2P para a implementação de hardware e o PIC 18F4620, da Microchip, para a implementação software, descritos à frente. Os factores que levaram a esta escolha compreenderam o baixo custo, fácil acesso e ser uma tecnologia conhecida (diminuindo o tempo de desenvolvimento). É de sublinhar que se poderia ter utilizado qualquer dispositivo que implemente o protocolo RS-232.

4.1.1 Xilinx Spartan-3 Starter Kit Board

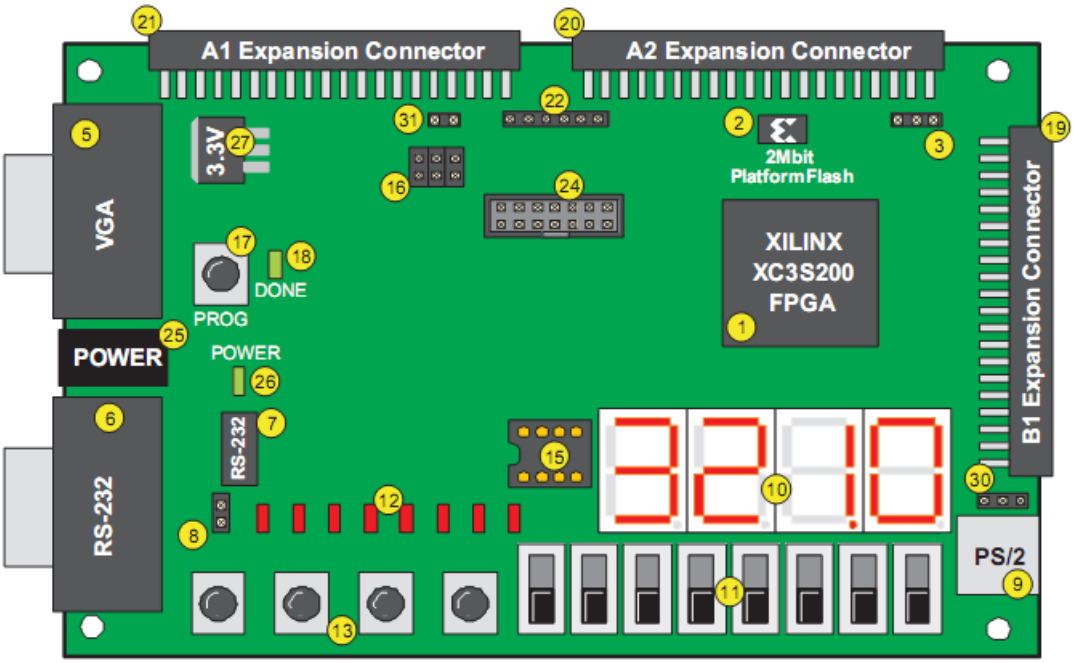


Figura 4.1 - Xilinx Spartan-3 Starter Kit Board (Vista de cima) [41].

A Figura 4.1, retirada de [41], mostra a placa do Kit da Xilinx, constituído por: uma FPGA Xilinx Spartan-3 (XC3S200 FT25); Plataforma flash XCF02S Xilinx; Duas memórias SRAM (256K x 16); Quatro botões de pressão; Oito interruptores; Oito LEDS; Quatro *displays 7-segmentos*; RS232; JTAG3; JTAG-IV; PS/2 (rato ou teclado); Porta VGA – 3 bits (8 cores); e três portos de expansão, possibilitando a interligação da FPGA com outros dispositivos. Para além da placa, o kit tem incluído um cabo de programação JTAG3; um transformador AC/DC; um guia de utilização; e o Software Xilinx ISE ® WebPack.

A FPGA incluída apresenta: 4320 células lógicas, dispostas em 480 blocos lógicos configuráveis; 200.000 portas lógicas; 3070 bits de memória RAM distribuída; 221184 bits de memória RAM em bloco; e 173 entradas/saídas (no máximo) [42].

4.1.2 Kit didáctico XUP VirtexTM – II Pro

O kit didáctico XUPV2P, representado através do diagrama de blocos da Figura 4.2, retirado de [6], é mais avançado que o atrás descrito, composto pela FPGA Xilinx Spartan-3, sendo constituído por: uma FPGA Virtex-II Pro XC2VP30; memória RAM e dois processadores PowerPC; porta Ethernet 10/100; encaixe *compact flash* para armazenamento de dados; porta de vídeo XSGA; *codec* de áudio; encaixe SATA para ligação de várias placas XUP; portas PS/2 e RS-232; quatro interruptores; cinco botões de pressão; quatro LEDs; três encaixes áudio (*Line-in, Microphone-in, Line-out*); e portos de expansão (de baixa e alta velocidade).

A FPGA possui: 30816 células lógicas, divididas em 13696 blocos lógicos;

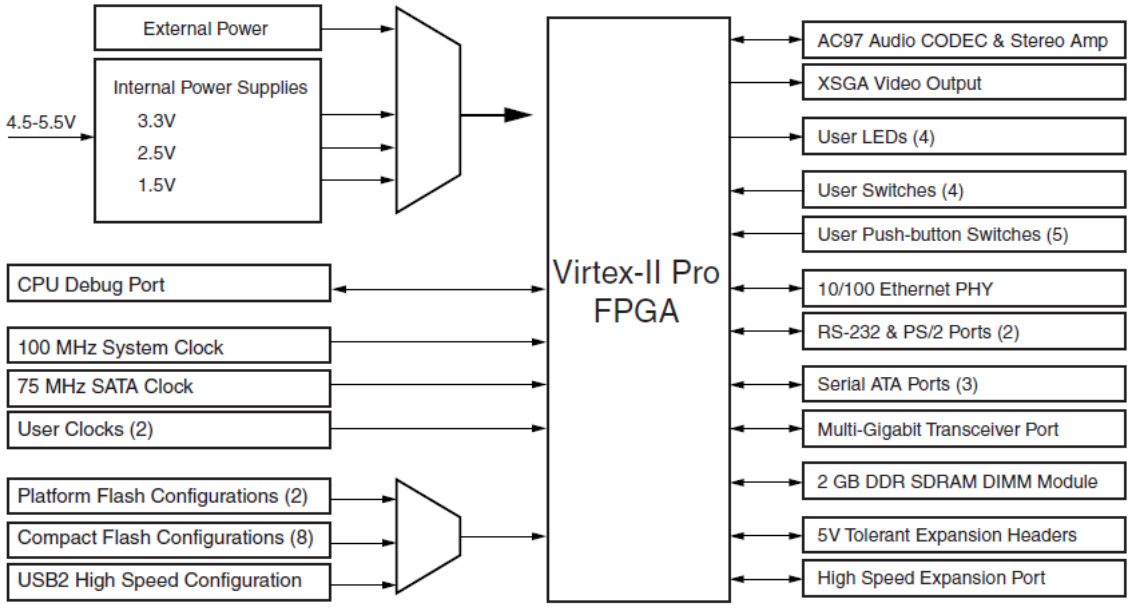


Figura 4.2 - Diagrama de blocos do kit XUP Virtex-II Pro [6]

438272 bits de memória RAM distribuída; 303552 bits de memória RAM em bloco; e 644 entrada/saída (no máximo) [43].

4.1.3 Microcontrolador PIC 18F4620 da Microchip

Os PIC são uma família de microcontroladores produzidos pela Microchip Technology Inc., também fabricante de semicondutores analógicos. Devido ao seu preço reduzido, notas de aplicação extensas e ferramentas de desenvolvimento gratuitas, tornaram-se muito populares entre projectistas e amadores.

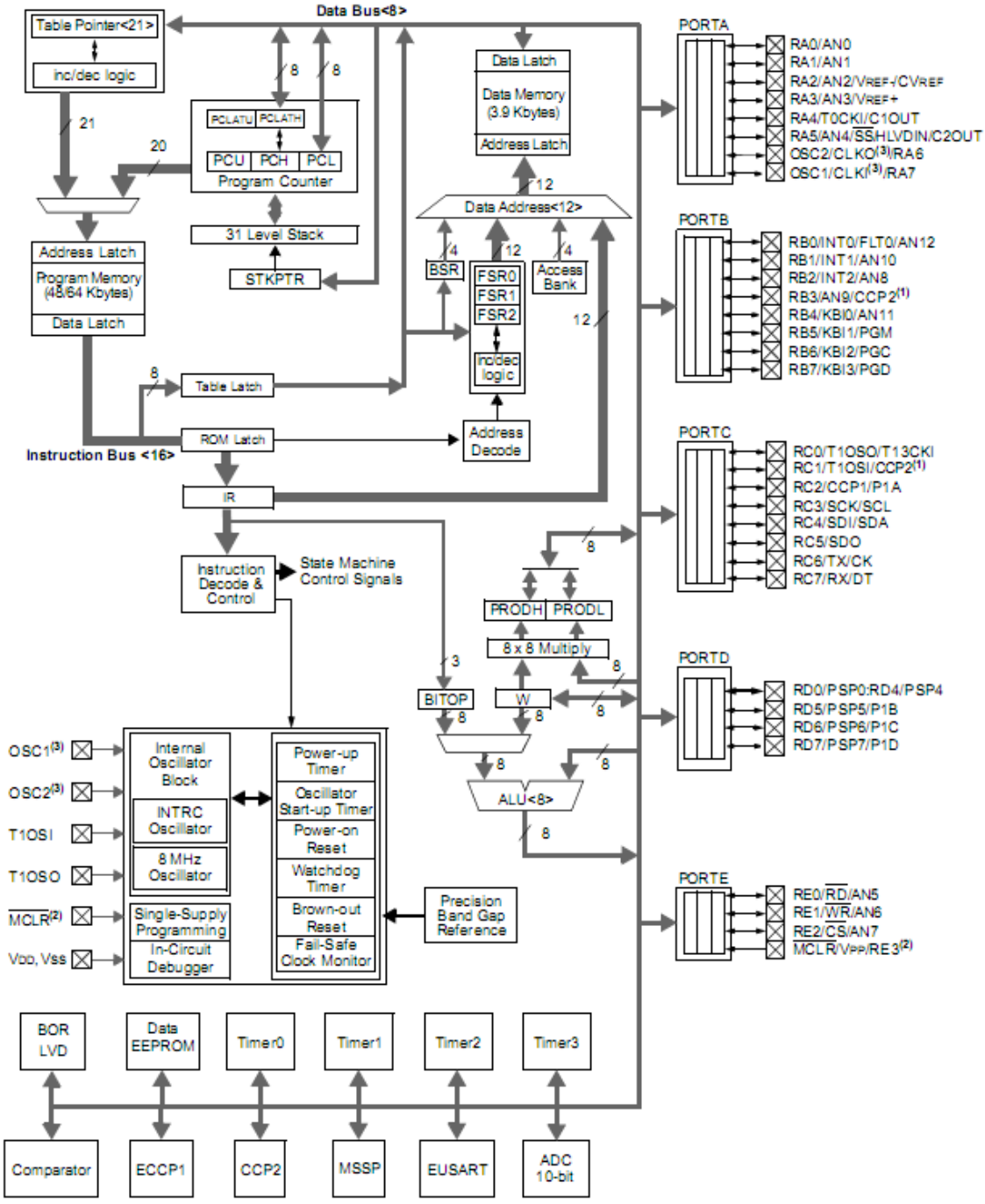
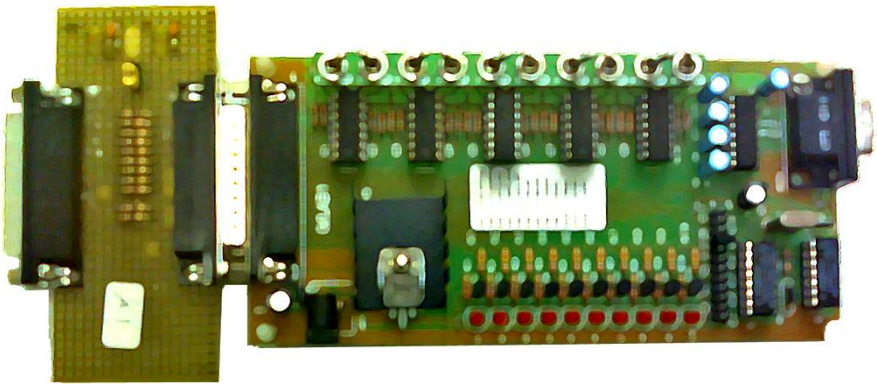


Figura 4.3 - Diagrama de blocos do PIC18F4620 (40/44 pinos) [1].

O PIC 18F4620 possui 64 KB de memória de programa flash, 4KB de memória RAM e 1KB de EEPROM, sem possibilidade de expansão. No circuito existem vários periféricos incluídos, como se pode ver na Figura 4.3 retirada de [1], tais como: temporizadores; comparadores; conversor analógico / digital e digital / analógico; UARTS; etc.

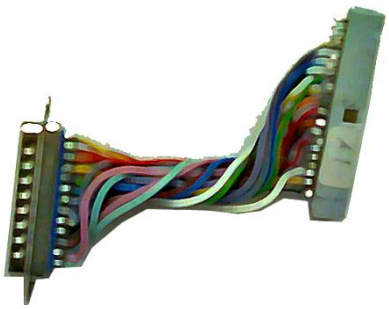
4.1.4 IMLAB1

Para se aproveitarem os portos de expansão das placas Xilinx, referidas atrás, foi utilizada uma placa produzida na faculdade (Figura 4.4) cuja designação é IMLAB1. A IMLAB1 é construída por: Dez LEDS; Dez interruptores; e uma porta RS-232, com respectivo transdutor. Para que não existissem problemas de tensão (a FPGA trabalha com valores de 3,3 V e a IMLAB1 com valores de 5 V), acrescentou-se uma placa com umas resistências, aproveitando-se para adicionar também alguns osciladores, a fim de se poder trabalhar com velocidades / fases de relógio diferentes, fazendo a interface entre o IMLAB1 e a FPGA.



a) Interface de ligação entre IMLAB1 e a FPGA

b) Placa IMLAB1



c) Cabo de ligação entre a FPGA e a placa IMLAB1

Figura 4.4 - Placa IMLAB1 desenvolvida na Faculdade de Ciências e Tecnologia, UNL.

4.2 Implementação

Como exemplos de aplicação foram considerados dois sistemas que, embora de complexidade relativa, academicamente justificam o uso de execução paralela.

Para cada exemplo são considerados três cenários de aplicação: (1) Ligação entre componentes numa só plataforma (hardware) sem recorrer à solução proposta; (2) Ligação entre componentes numa só plataforma (hardware e software) recorrendo à solução descrita no capítulo anterior; e (3) Ligação entre componentes em plataformas heterogéneas.

4.2.1 Exemplo I – Quatro carros sincronizados

Como exemplo de aplicação de um sistema de automação, em que é benéfica a execução concorrente de submodelos resultantes da partição do modelo, recorrer-se-á ao exemplo inicialmente introduzido em [44], onde o sistema a controlar é composto por quatro carros que se movimentam entre dois pontos *Ai* e *Bi* sincronizando-se o seu movimento através dos botões de *Go* e *Back*. O sistema é ilustrado na Figura 4.5.

Figura 4.5 - Exemplo de aplicação – 4 carros sincronizados.

Sendo assim, o controlador do sistema tem como sinais de entrada A1, A2, A3, A4, sinalizando os pontos de início de cada trajectória, B1, B2, B3 e B4 instalados no fim do percurso, e os botões de sincronização dos movimentos Go e Back, dando ordem para os carros iniciarem o seu movimento em direcção de B ou A respectivamente. Como sinais de saída considera-se M1_DB, M2_DB, M3_DB, M4_DB indicando que o carro está em movimento em direcção de B e M1_DA, M2_DA, M3_DA, M4_DA indicando o movimento em direcção oposta.

Considerando as redes de Petri IOPT [9] como formalismo de modelação para o comportamento do controlador do sistema, obteve-se a rede apresentada na

Figura 4.6. Para não sobrecarregar a figura, os sinais de saída associados à actuação dos motores, apesar de definidos, não estão visíveis no modelo.

Tendo em conta que o objectivo é obter um controlador para cada carro, este modelo foi decomposto em quatro submodelos que podem ser considerados como o modelo do controlador de cada um dos carros. Para esse efeito utilizou-se a operação Net Splitting [29] e a ferramenta de SPLIT. Na Figura 4.6 estão assinalados os nós por onde se deve partir o modelo para obter a decomposição desejada. A sua selecção é realizada tendo em conta o modelo inicial e um objectivo específico (neste caso obter um controlador para cada um dos carros do sistema).

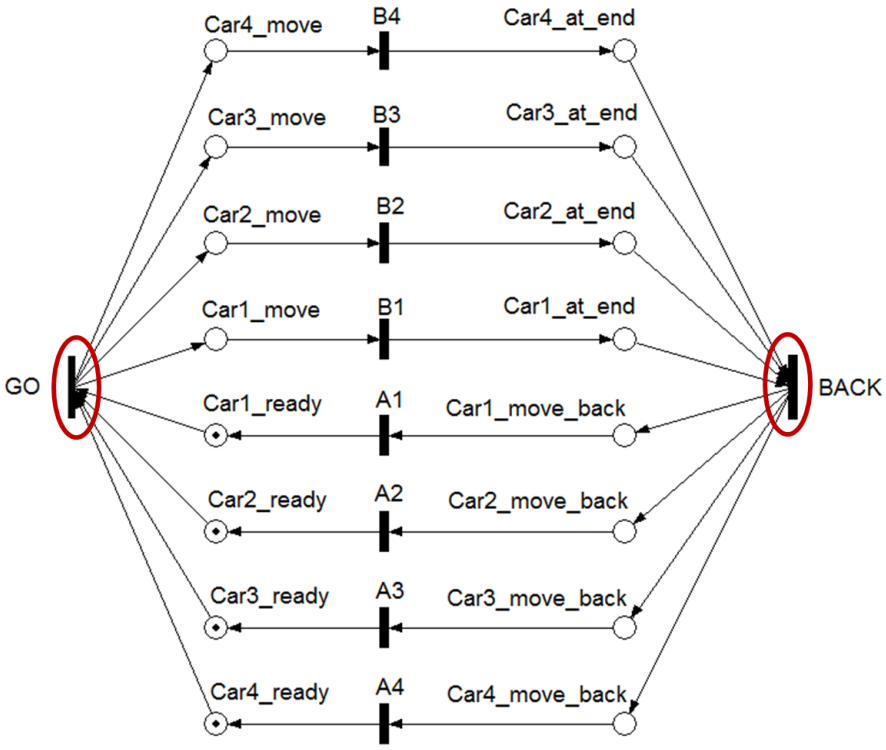


Figura 4.6 - Modelo RdP do controlador do sistema, com o conjunto de corte assinalado (Exemplo I).

Visto o formalismo de modelação utilizado ser o das redes de Petri IOPT, serão incluídos os eventos internos gerados nos modelos resultantes da partição do modelo. Estes eventos gerados são associados aos canais de comunicação. Na interligação entre os submodelos considera-se a interligação dos eventos de saída e entrada com nomes semelhantes, como ilustrado na Figura 4.7, onde os eventos de nome “inevent???” e “outevent???” garantem a comunicação entre as transições “???_master” e “???_slave”.

A ferramenta SPLIT, que executa a decomposição do modelo, gera estes eventos de comunicação de forma a permitir identificação automática para efeitos de

interligação. Na Figura 4.7 estão representados os módulos que vão ser implementados em cada controlador. Neste modelo estão incluídos os eventos de entrada e de saída gerados, bem como a sua interligação e os sinais de entrada e saída que comunicam com o mundo exterior.

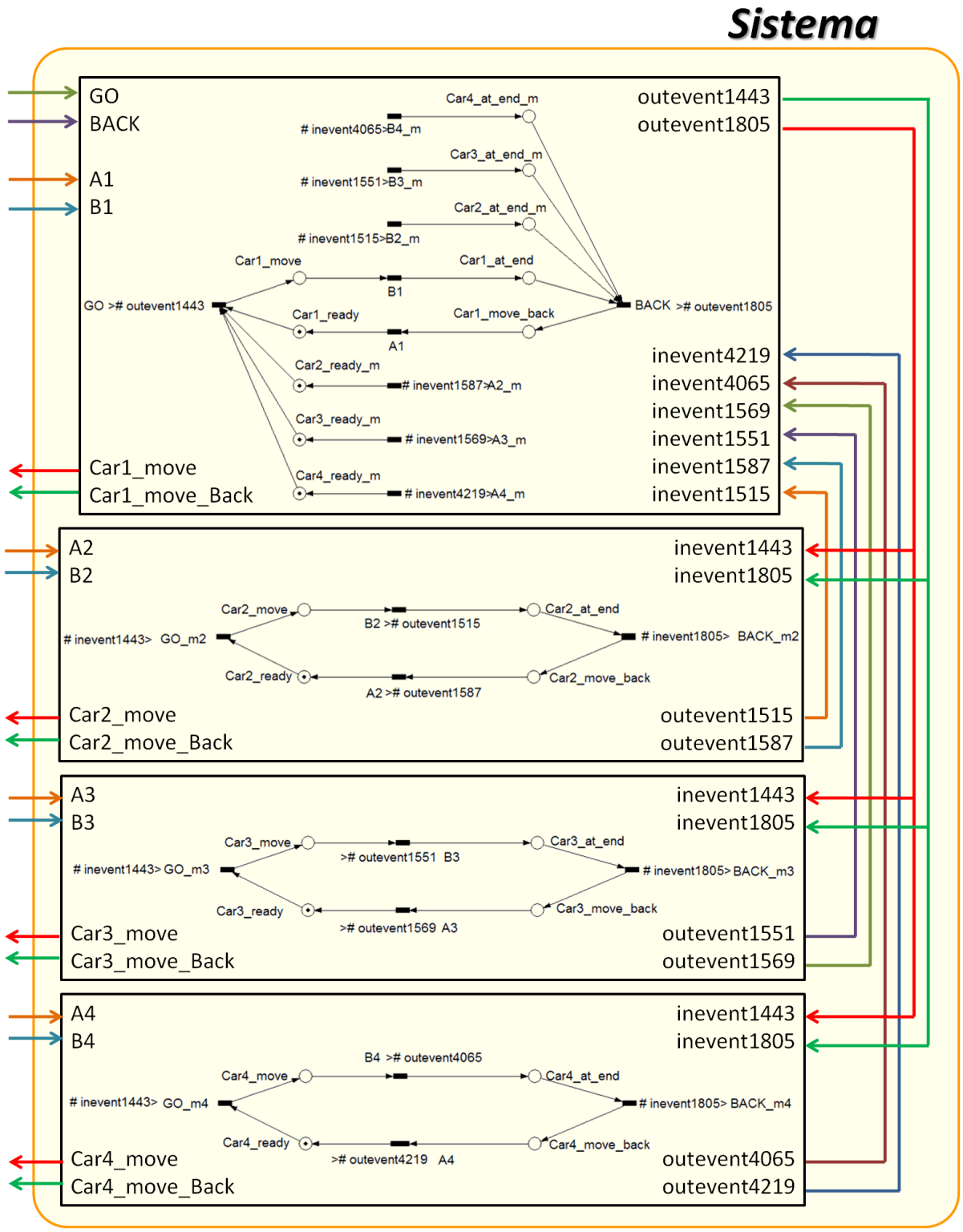


Figura 4.7 - Modelos dos controladores dos vários carros.

A ferramenta SPLIT lê o ficheiro PNML do modelo inicial, e gera os submodelos e os ficheiros PNML associados para cada módulo.

Como se pode verificar na Figura 4.7, o sistema tem os sinais de entrada A1, A2, A3, A4, B1, B2, B3 e B4 comunicando directamente ao respectivo módulo. Os sinais GO e BACK estão associados ao módulo do carro1, pois foi este o definido como *master*. Ou seja, sempre que ocorrer um evento associado ao GO, no módulo do carro1 é gerado um evento de saída (outevent1443) que ligará aos módulos do carros2, carro3 e carro4 através de inevent1443. O processo é idêntico para o sinal BACK, mas tendo como evento de saída o outevent1805 com correspondência ao inevent1805 nos módulos dos outros carros. Com o mesmo princípio, se o módulo do carro1 "informou" os restantes módulos de GO e BACK terá que ser informado de A2, A3, A4, B2, B3 e B4. A Tabela 4.1 mostra os eventos de entrada e saída a cada um desses sinais. Os lugares Car#_move e Car#_move_back têm os sinais M#_DB e M#_DA de saída, nos módulos respectivos, associados para indicar o movimento.

Tabela 4.1 - Correspondência entre os eventos de saída de cada carro com os eventos de entrada do carro1

	Sinal	*Evento de saída*	*Evento de entrada*	
Carro2	A2	outevent1587	inevent1587	**Carro1**
	B2	outevent1515	inevent1515	
Carro3	A3	outevent1569	inevent1569	
	B3	outevent1551	inevent1551	
Carro4	A4	outevent4219	inevent4219	
	B4	outevent4065	inevent4065	

4.2.1.1 Ligação entre componentes numa só plataforma sem recorrer à solução proposta

Para se poderem concluir no próximo capítulo os resultados da solução proposta, primeiro será implementado o sistema tal como é mostrado na Figura 4.7. Como se pode verificar, existem 4 módulos e os eventos / sinais são ligados entre eles de forma *ponto-a-ponto*. Utilizou-se a ferramenta realizada em [28], de forma a acelerar o processo, e obtiveram-se os resultados mostrados na Tabela 4.2.

A área de silício, estimada através dos recursos usados, e o consumo de energia são mostrados na Tabela 4.2 e Figura 4.8, respectivamente, sendo, tal como referido no capítulo dois, duas das métricas frequentemente utilizadas na literatura aquando da análise deste tipo de soluções.

Tabela 4.2 - Sumário da utilização do dispositivo, usando o cenário de aplicação depois de aplicar a ferramenta SPLIT, ligações ponto-a-ponto (Exemplo I).

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	28	3,840	1%
Number of 4 input LUTs	41	3,840	1%
Number of occupied Slices	28	1,920	1%
Number of Slices containing only related logic	28	28	100%
Number of Slices containing unrelated logic	0	28	0%
Total Number of 4 input LUTs	41	3,840	1%
Number of bonded IOBs	16	173	9%
IOB Flip Flops	6		
Number of GCLKs	1	8	12%
Total equivalent gate count for design	521		
Additional JTAG gate count for IOBs	768		

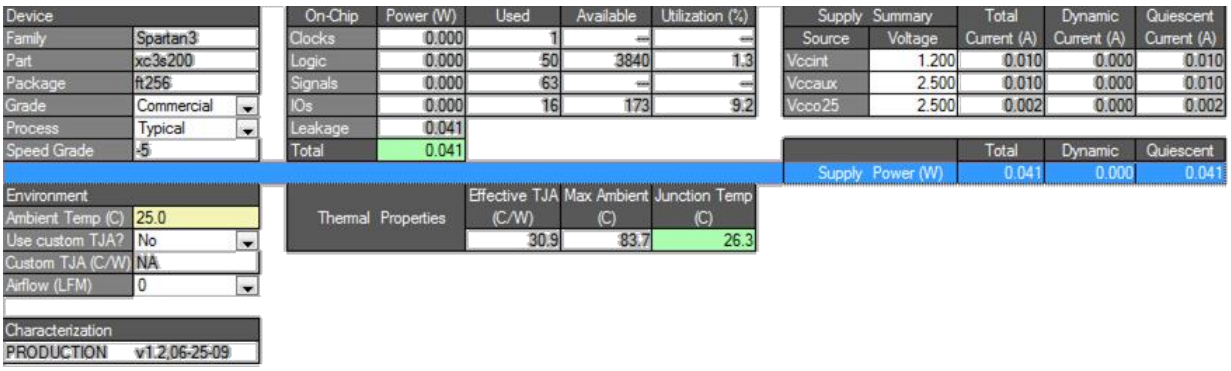


Figura 4.8 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando o cenário de aplicação depois de aplicar a ferramenta SPLIT, ligações ponto-a-ponto (Exemplo I).

O sistema foi implementado na *Xilinx Spartan-3 Starter Kit Board*, utilizando os interruptores para simular os sensores A# e B# de todos os carros, os LEDS para os actuadores dos motores e, finalmente, nos botões de pressão colocaram-se os botões de sincronismo, para além do *Reset* do sistema.

O resultado alcançado foi o esperado, mostrando o sucesso da modelação, mesmo em sistemas distribuídos.

4.2.1.2 Ligação entre componentes numa só plataforma

Uma vez obtidos os ficheiros PNML de cada submodelo, são analisados os sinais e/ou eventos de cada um dos submodelos possibilitando definir o endereço e a topologia da rede de interligação. Os algoritmos e regras descritos em 3.3.1, que nos permitem fazê-lo, foram implementados, com recurso à linguagem de programação Java, utilizando a distribuição gratuita e *open-source* NetBeans IDE

(*Integrated Development Environment*) 6.9.1. O autor, após transmitir o número de módulos existentes, fornecerá o número de mensagens entre nós (Figura 4.9) e os ritmos de transmissão (Figura 4.10).

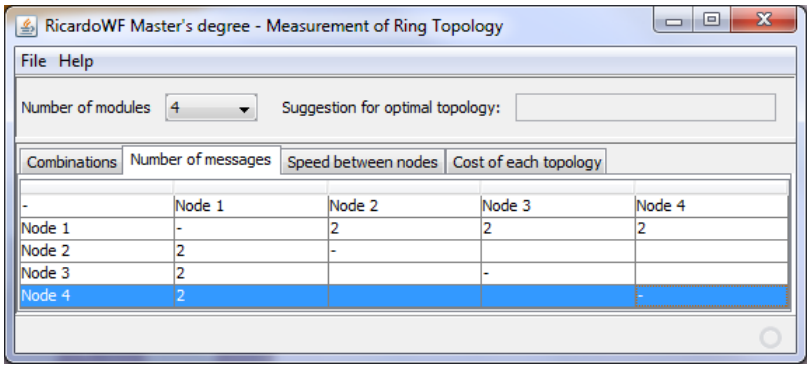


Figura 4.9 - Inserção do número de mensagens trocadas entre nós (Exemplo I)

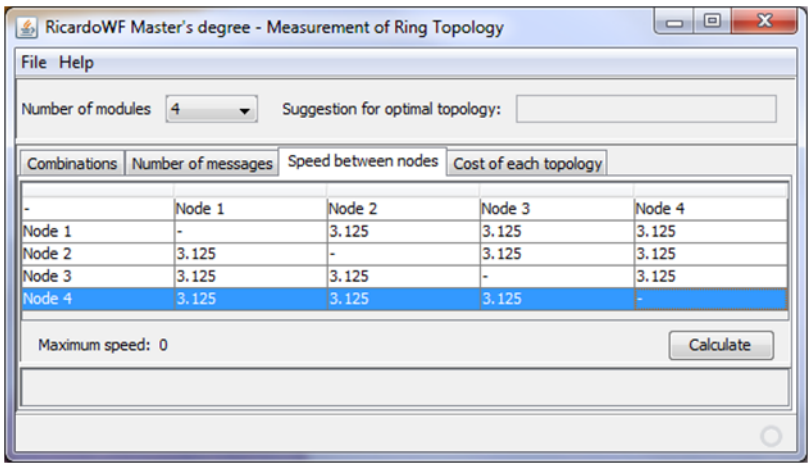
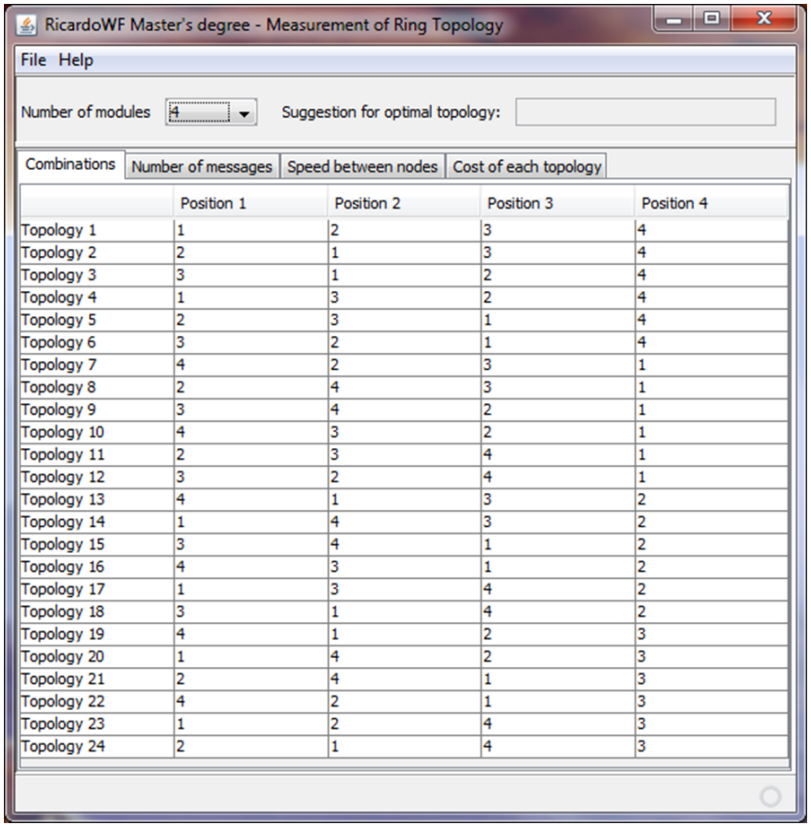


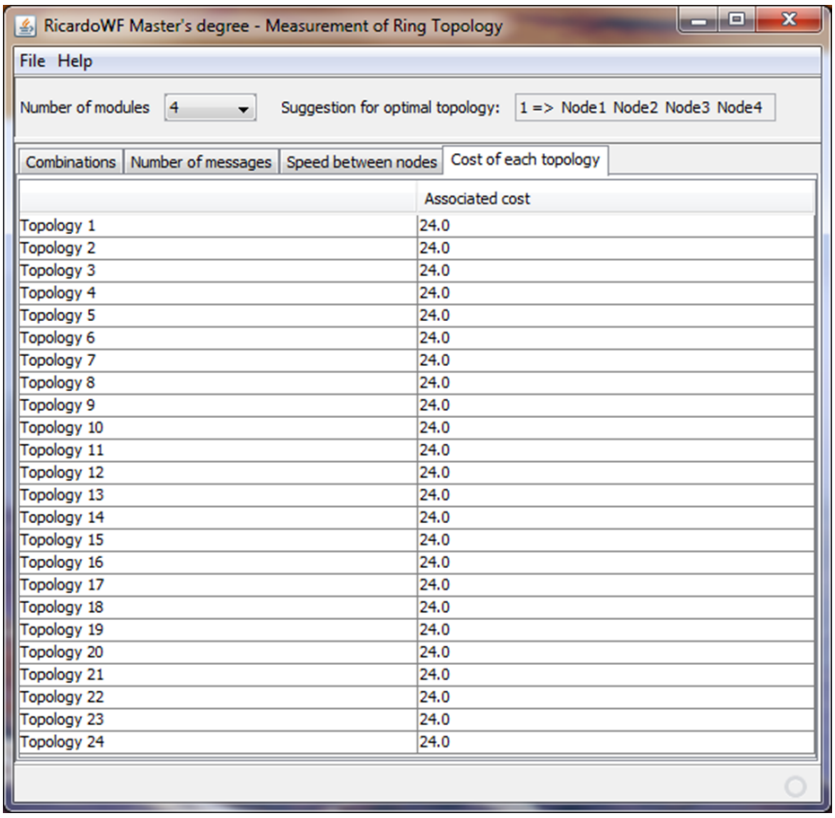
Figura 4.10 - Inserção das taxas de transmissão entre nós

Após a inserção dos dados, e depois de se terem dividido todas as taxas de transmissão pela maior taxa, conforme o algoritmo apresentado na Figura 3.6, a aplicação calcula o custo para cada uma das topologias possíveis (Figura 4.11), tal como é apresentado na Figura 4.12. O custo obtido para cada uma das topologias foi sempre de 24 porque, com excepção do controlador do carro1, estima-se que todos os controladores enviem e recebam 2 mensagens em iguais condições. Como se está num anel, qualquer que seja a posição do carro1 em relação aos restantes, o custo será o mesmo. A escolha recai sempre na primeira topologia que tenha o menor custo, sendo que, para este exemplo recai na primeira escolha ficando-se com a seguinte topologia: *Nó1 – Nó2 – Nó3 – Nó4*.



	Position 1	Position 2	Position 3	Position 4
Topology 1	1	2	3	4
Topology 2	2	1	3	4
Topology 3	3	1	2	4
Topology 4	1	3	2	4
Topology 5	2	3	1	4
Topology 6	3	2	1	4
Topology 7	4	2	3	1
Topology 8	2	4	3	1
Topology 9	3	4	2	1
Topology 10	4	3	2	1
Topology 11	2	3	4	1
Topology 12	3	2	4	1
Topology 13	4	1	3	2
Topology 14	1	4	3	2
Topology 15	3	4	1	2
Topology 16	4	3	1	2
Topology 17	1	3	4	2
Topology 18	3	1	4	2
Topology 19	4	1	2	3
Topology 20	1	4	2	3
Topology 21	2	4	1	3
Topology 22	4	2	1	3
Topology 23	1	2	4	3
Topology 24	2	1	4	3

Figura 4.11 - Posição de cada nó de rede na topologia em anel dada pela aplicação



	Associated cost
Topology 1	24.0
Topology 2	24.0
Topology 3	24.0
Topology 4	24.0
Topology 5	24.0
Topology 6	24.0
Topology 7	24.0
Topology 8	24.0
Topology 9	24.0
Topology 10	24.0
Topology 11	24.0
Topology 12	24.0
Topology 13	24.0
Topology 14	24.0
Topology 15	24.0
Topology 16	24.0
Topology 17	24.0
Topology 18	24.0
Topology 19	24.0
Topology 20	24.0
Topology 21	24.0
Topology 22	24.0
Topology 23	24.0
Topology 24	24.0

Figura 4.12 - Custo associado a cada topologia (Exemplo I)

Tendo em conta o resultado obtido, foi atribuído o endereço de rede #00 ao carro1, #01 ao carro2, #10 para o carro3 e #11 para o carro4.

Também da análise dos ficheiros PNML, foram considerados os sinais e/ou eventos de entrada de cada carro que comuniquem com outro(s) carro(s) e são codificados, como mostra a Tabela 4.3, permitindo posteriormente a composição de mensagens (Figura 4.13).

Tabela 4.3 - Codificação dos eventos de entrada de cada carro

Carro1 - #00

Evento	Código
Go	-
Back	-
A1	-
B1	-
inevent1515	000
inevent1587	001
inevent1569	010
inevent1551	011
Inevent4065	100
Inevent4219	101

Carro2 - #01

Evento	Código
A2	-
B2	-
inevent1805	000
inevent1443	001

Carro3 - #10

Evento	Código
A3	-
B3	-
inevent1805	000
inevent1443	001

Carro4 - #11

Evento	Código
A3	-
B3	-
inevent1805	000
inevent1443	001

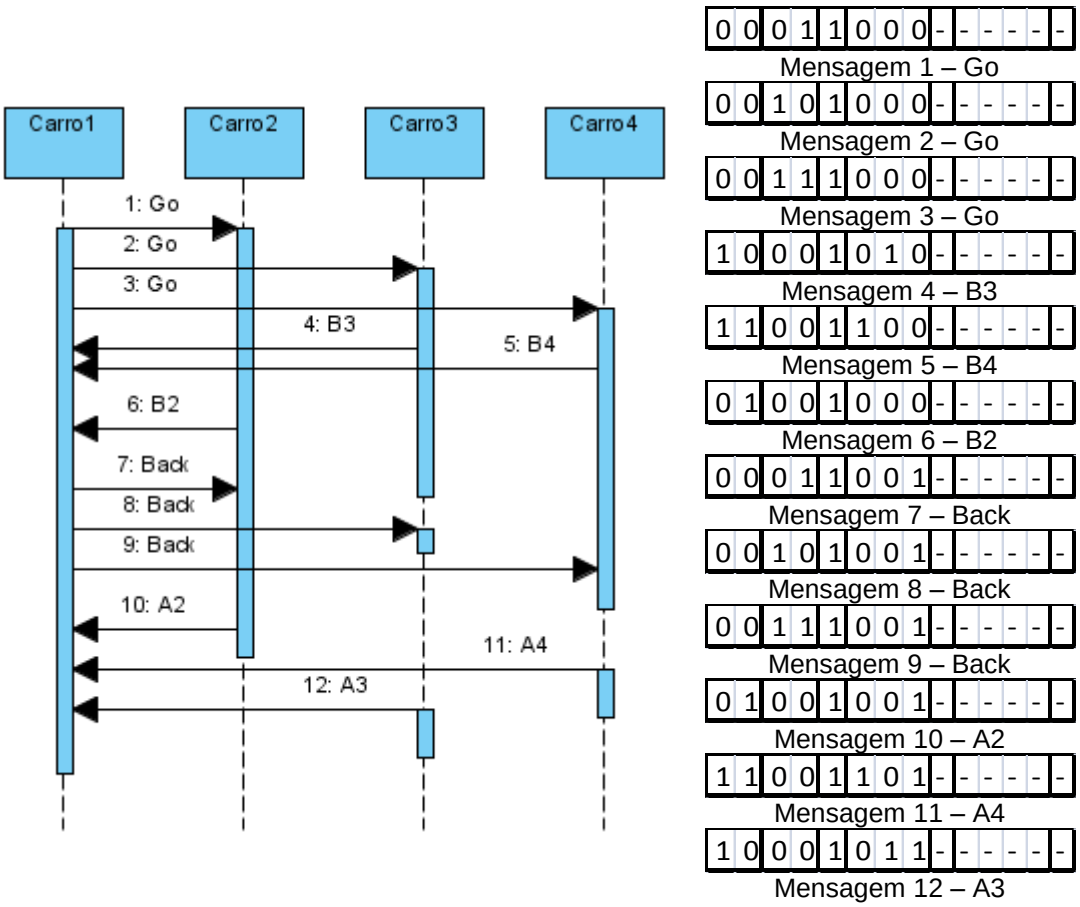


Figura 4.13 - Mensagens a circular na rede (Exemplo I)

Com base em todos estes passos já se pode descrever o hardware / software que se irá utilizar. Através da descrição das plataformas, feita pelo autor, sabe-se a velocidade dos osciladores onde cada carro será implementado. Os módulos hardware, neste cenário, foram todos implementados numa só plataforma (no kit SPARTAN-III e no kit XUPV2P - Figura 4.14). No kit SPARTAN-III, e apesar de todos os módulos terem sido lá implementados, foram considerados, com recurso do IMLAB1, velocidade de relógio diferentes (Carro1 – 50 MHz; Carro2 – 32,7 MHz; Carro3 – 24 MHz; e Carro4 – 10 MHz). Como os osciladores do IMLAB1 não se puderam integrar no kit XUPV2P através dos portos de expansão, foram utilizados os dois osciladores do kit e invertidos de forma a obter quatro sinais (Carro1 – 32 MHz; Carro2 – 32 MHz desfasado; Carro3 – 100 MHz; e Carro4 – 100 MHz desfasado).

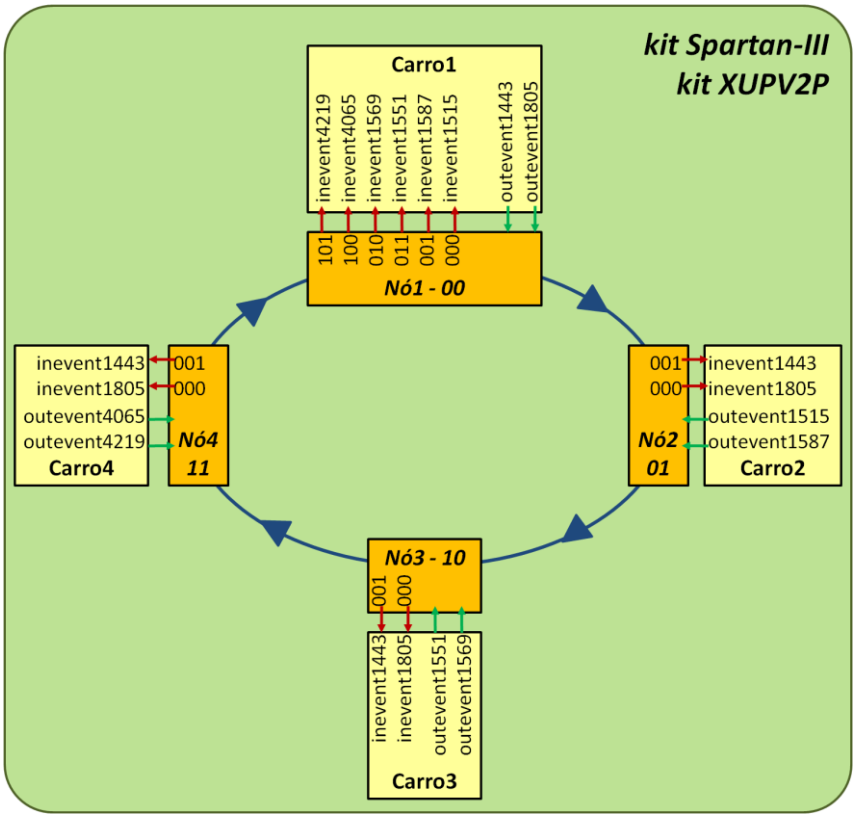


Figura 4.14 - Implementação hardware numa só plataforma (Exemplo I)

Começar-se-á por modular o sinal que irá controlar a taxa de transmissão das UART (no *template* BAUDrate.vhdl). Considerando que se tem uma transmissão 3,125 Mbps, como referido na Figura 4.10, então $\mathrm{EN\_16\_X\_BAUD} = 16 \times 3{,}125\ \mathrm{MHz} = 50\ MHz$, o que implica ter no kit Spartan-III um

$$baud\_count\_max = \frac{50\ MHz\ (velocidade\ de\ relógio)}{50\ MHz\ (EN\_16\_X\_BAUD)} = 1.$$

e no kit XUPV2P um

$$baud\_count\_max = \frac{100\ MHz\ (velocidade\ de\ relógio)}{50\ MHz\ (EN\_16\_X\_BAUD)} = 2.$$

Nos restantes *templates* descritos no capítulo anterior, e tendo em conta as codificações já encontradas, a Tabela 4.4 mostra os valores que foram atribuídos aos sinais genéricos.

Tabela 4.4 - Atribuição dos valores dos sinais genéricos (Exemplo I)

Sinal genérico	Carro1	Carro2	Carro3	Carro4
MaxLenght	3			
DimMsg	14			
Dim_Node	2			
Dim_IDsignal	3			
Dim_Signal	1			
DIm_CRC	1			
CurrentNode	00	01	10	11
DestinationNode1	01	00	00	00
Node1SinalID	000	000	010	100
DestinationNode2	10	00	00	00
Node2SinalID	000	001	011	101
DestinationNode3	11	***Não existe***		
Node3SinalID	000			
DestinationNode4	01			
Node4SinalID	001			
DestinationNode5	10			
Node5SinalID	001			
DestinationNode6	11			
Node6SinalID	001			
Num_signalsSend	3	1	1	1
Buffer_Lenght	8			
Vector_width	13			
Num_signalsReceiv	6	2	2	2
Num_sinais	6	2	2	2

O tamanho dos *buffers* é de oito (Buffer_Lenght = 8 e MaxLenght = 3) unicamente por uma questão de segurança obtendo-se assim a garantia de que não se perderia informação. Tendo em conta que cada modelo pode receber dois eventos ao mesmo tempo (no máximo) e que esse processamento é mais rápido que a transmissão / recepção completa de uma mensagem (aproximadamente 160 ciclos de relógio - 10 bits X 16 ciclos de relógio). No pior cenário, seriam precisas 4 posições de memória (uma para cada evento, uma para uma mensagem recebida e outra de reserva).

Com o intuito de se verificar, na prática, o tempo que demora entre os eventos de saída e a recepção / processamento da mensagem por eles provocadas, foram verificados os instantes de tempo, descritos na Tabela 4.5, para a execução do exemplo. O intervalo de tempo entre os eventos, bem como a sua duração não é realista, ou seja, possivelmente entre a posição A e B um carro demoraria minutos, no entanto, foram considerados valores bem inferiores, de forma a tornar mais fácil a cronometração. Contudo, a duração da transferência da mensagem e seu processamento não é afectado, sendo o tempo determinado realista.

Tabela 4.5 - Instantes de tempo de cada sinal do exemplo da Figura 4.13, obtidos através do ModelSim (Exemplo I)

Sinal	*Instante de tempo [ms]*
Go	0.0000300
outevent1443	0.0000300
Carro2_inevent1443	0.0067115
Car2Moves	0.0067425
Carro3_inevent1443	0.0196770
Car3Moves	0.0197190
Carro4_inevent1443	0.0326500
Car4Moves	0.0327500
B3	2.0000300
outevent1551	2.0000610
inevent1551	2.0132700
B4	3.0000300
outevent4065	3.0000500
inevent4065	3.0067900
B2	4.0000300
outevent1515	4.0000385
inevent1515	4.0197700
Back	5.0000300
outevent1805	5.0000300
Carro2_inevent1805	5.0067325
Car2MovesBack	5.0067635
Carro3_inevent1805	5.0196510
Car3MovesBack	5.0196930
Carro4_inevent1805	5.0326500
Car4MovesBack	5.0327500
A2	6.0000300
outevent1587	6.0000345
inevent1587	6.0197700
A4	8.0000300
outevent4219	8.0000500
inevent4219	8.0067900
A3	9.0000300
outevent1569	9.0000330
inevent1569	9.0132500

4.2.1.2.1 Recursos utilizados e consumos na plataforma Spartan-III

Tal como anteriormente, o sistema foi implementado na *Xilinx Spartan-3 Starter Kit Board*, utilizando os interruptores para simular os sensores A# e B# de todos os

carros, os LEDS para os actuadores dos motores e, finalmente, nos botões de pressão colocaram-se os botões de sincronismo.

Os recursos usados e o consumo de energia são mostrados na Tabela 4.6 e na Figura 4.15, respectivamente.

Tabela 4.6 - Sumário da utilização do dispositivo utilizando a solução proposta só na plataforma Spartan-III (Exemplo I).

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,622	3,840	42%
Number of 4 input LUTs	2,079	3,840	54%
Number of occupied Slices	1,243	1,920	64%
Number of Slices containing only related logic	1,243	1,243	100%
Number of Slices containing unrelated logic	0	1,243	0%
Total Number of 4 input LUTs	2,143	3,840	55%
Number used as logic	1,939		
Number used as a route-thru	64		
Number used as Shift registers	140		
Number of bonded IOBs	16	173	9%
IOB Flip Flops	6		
Number of BUFGMUXs	1	8	12%
Average Fanout of Non-Clock Nets	4.44		

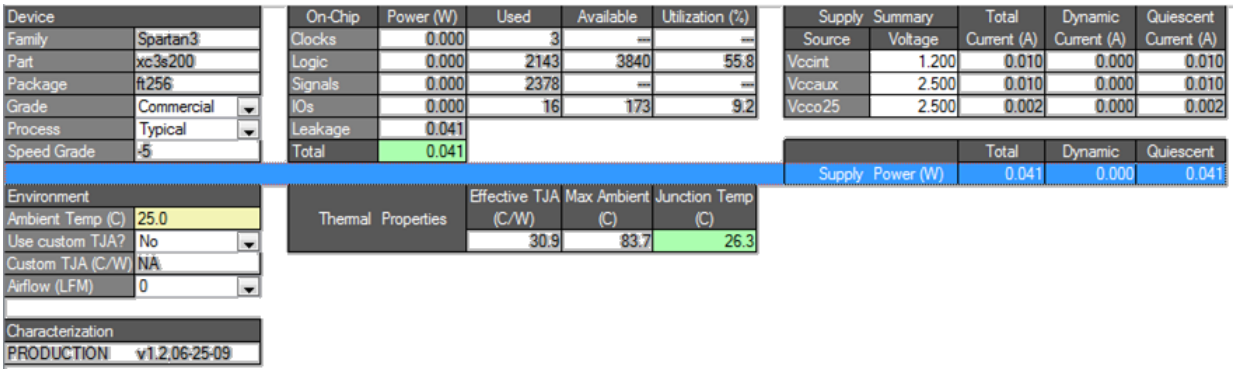


Figura 4.15 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta só na plataforma Spartan-III (Exemplo I).

4.2.1.2.2 Recursos utilizados e consumos na plataforma XUPV2P

Nesta plataforma, e como só existem disponíveis quatro interruptores e cinco botões de pressão, foram utilizados os interruptores do IMLAB1 para simular os sensores e os botões de sincronismo. Pela mesma razão, não foram utilizados os LEDs da plataforma e sim os do IMLAB1.

A Tabela 4.7 e a Figura 4.16 mostram os recursos utilizados e o consumo de energia.

Tabela 4.7 -Sumário da utilização do dispositivo utilizando a solução proposta só na plataforma XUPV2P (Exemplo I).

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,624	27,392	5%
Number of 4 input LUTs	2,006	27,392	7%
Logic Distribution			
Number of occupied Slices	1,214	13,696	8%
Number of Slices containing only related logic	1,214	1,214	100%
Number of Slices containing unrelated logic	0	1,214	0%
Total Number of 4 input LUTs	2,070	27,392	7%
Number used as logic	1,866		
Number used as a route-thru	64		
Number used as Shift registers	140		
Number of bonded IOBs			
Number of bonded	16	556	2%
Number of BUFGMUXs	1	16	6%

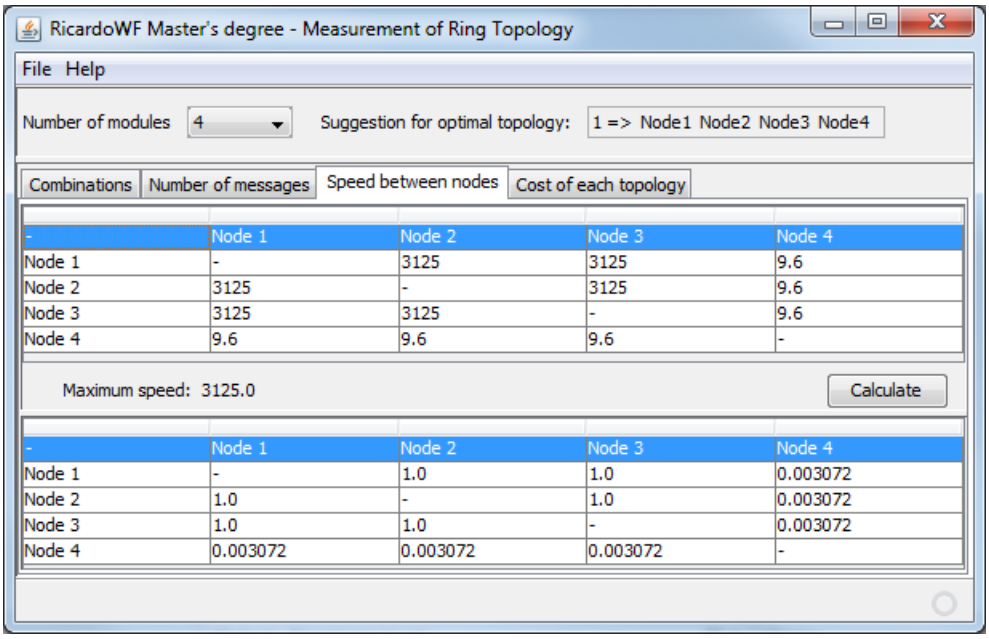
Name	Value	Used	Total Available	Utilization (%)
Clocks	0.00952 (W)	3	—	—
Logic	0.00040 (W)	2070	27392	7.6
Signals	0.00427 (W)	3703	—	—
IOs	0.00143 (W)	16	588	2.7
Total Quiescent Power	**0.10313 (W)**			
Total Dynamic Power	**0.01562 (W)**			
Total Power	**0.11874 (W)**			
Junction Temp	**25.0 (degrees C)**			

Figura 4.16 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta só na plataforma XUPV2P (Exemplo I)

4.2.1.3 Ligação entre componentes em plataformas heterogéneas

Para aplicar o exemplo em plataformas heterogéneas, os ficheiros PNML são analisados da mesma forma que foram em 4.2.1.2. Serão consideradas três hipóteses de interligação dos módulos: (1) Carro1, Carro2 e Carro3 implementados no Kit Spartan-III e o Carro4 num microcontrolador PIC18F4620; (2) Em vez do kit Spartan-III, os carros um, dois e três são implementados no kit XUPV2C e o Carro4 mantém-se no PIC18F4620; e (3) ter cada um dos carros implementados num PIC18F4620. Relativamente a 4.2.1.2, a única diferença foi a velocidade de transmissão entre os módulos dos microcontroladores. Foi considerada uma taxa de transmissão de 9600 bps. Tanto na terceira hipótese, onde se manteve a mesma

velocidade entre todos os nós, como nas duas primeiras hipóteses foi sugerida (a Figura 4.17 mostra o resultado obtido após inserção das velocidades para as duas primeiras hipóteses) a mesma topologia.



-	Node 1	Node 2	Node 3	Node 4
Node 1	-	3125	3125	9.6
Node 2	3125	-	3125	9.6
Node 3	3125	3125	-	9.6
Node 4	9.6	9.6	9.6	-

-	Node 1	Node 2	Node 3	Node 4
Node 1	-	1.0	1.0	0.003072
Node 2	1.0	-	1.0	0.003072
Node 3	1.0	1.0	-	0.003072
Node 4	0.003072	0.003072	0.003072	-

Figura 4.17 - Inserção das velocidades das duas primeiras hipóteses.

Como se está perante os mesmos modelos e foi escolhida a mesma topologia, as codificações dos sinais / eventos mantêm-se os da Tabela 4.3, conservando as mensagens mostradas na Figura 4.13.

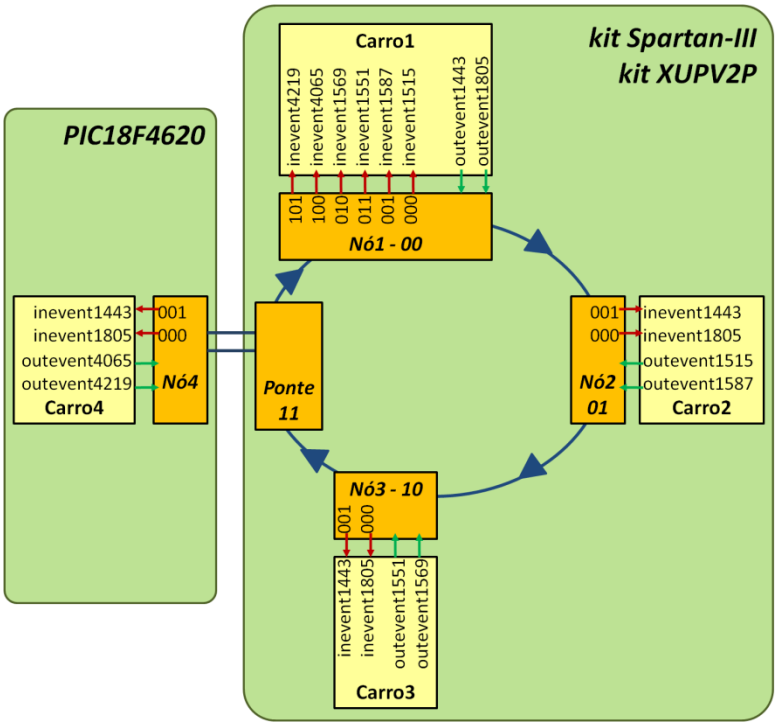


Figura 4.18 – Implementação em plataformas heterogéneas (entre kits Xilinx e microcontrolador - Exemplo I)

A Figura 4.18 e a Figura 4.19 ilustram como é feita a interligação entre as plataformas heterogéneas para as três hipóteses consideradas.

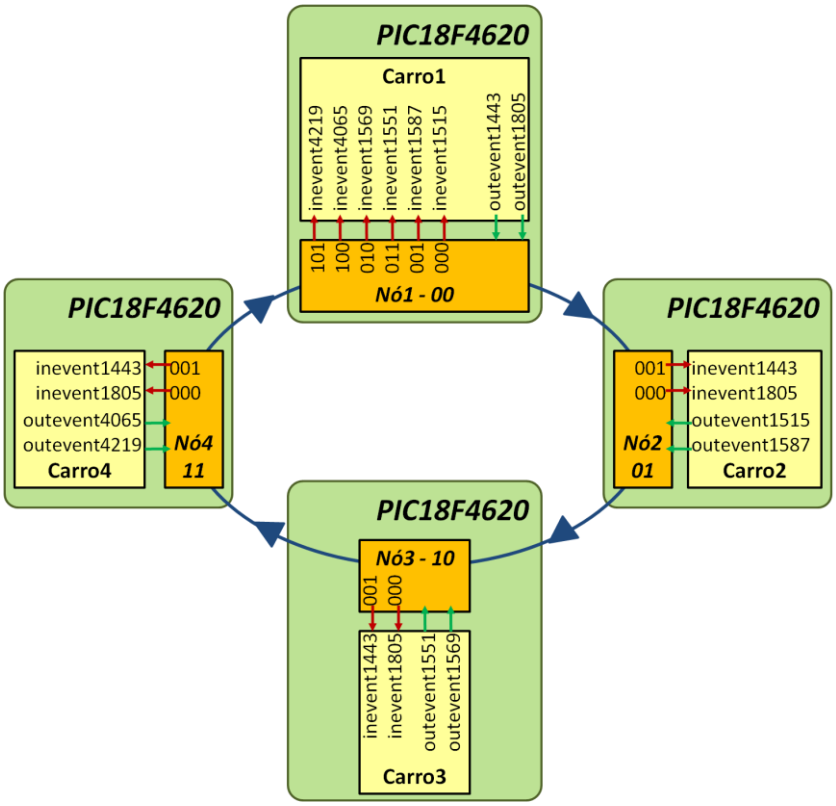


Figura 4.19 – Implementação em plataformas heterogéneas (entre quatro microcontrolador – Exemplo I)

No que diz respeito à descrição de hardware a utilizar, quase todos os sinais genéricos mantêm os mesmos valores (apresentados na Tabela 4.4) com excepção do sinal *baund_count_max* de envio do nó do Carro3 e recepção do nó do Carro1. Estes, como a taxa passou a ser de 9600 Hz, serão dados por: $\mathrm{EN\_16\_X\_BAUD} = 16 \times 9600\ \mathrm{Hz} = 153{,}6\ KHz$, o que implica ter no kit Spartan-III um

$$baud\_count\_max = \frac{50\ MHz\ (velocidade\ de\ relógio)}{153{,}6\ KHz\ (EN\_16\_X\_BAUD)} = 326.$$

e no kit XUPV2P um

$$baud\_count\_max = \frac{100\ MHz\ (velocidade\ de\ relógio)}{153{,}6\ KHz\ (EN\_16\_X\_BAUD)} = 651.$$

Em software foi configurada a UART do microcontrolador com recurso à função *OpensUSART* fornecida pelo compilador C18, da Microchip, descrita em [45]. O *interrupt* apenas foi considerado para a recepção; oito bits, modo assíncrono num modo contínuo de recepção. O oscilador utilizado é de 4.9152 MHz, que será dado pelo autor, e tal como referido em [1], ou em [45], a variável *spbrg* (segundo argumento da função *OpenUSART*) é dada por:

$$spbrg = \frac{\frac{4{,}9152 \times 10^6 \ \ (velocidade\ do\ oscilaor)}{9600 \ \ (ritmo\ desejado)}}{16} - 1 = 31$$

4.2.1.3.1 Recursos utilizados em cada um dos carros implementados num PIC18F4620

Para a implementação deste cenário foram utilizadas duas *breadboards* e uma placa de testes de circuitos TTLs, com interruptores e leds, tipicamente disponível em laboratórios de sistemas digitais. Para os interruptores (sensores / botões de sincronização do modelo inicial) recorreu-se ao registo PORTB do microcontrolador e para os LEDS (actuadores do modelo) foi utilizado o registo PORTD.

A Figura 4.20 mostra os recursos utilizados por cada PIC18F4620 para a execução do sistema. A medição é dada pelo ambiente de desenvolvimento, da Microchip, MPLAB.

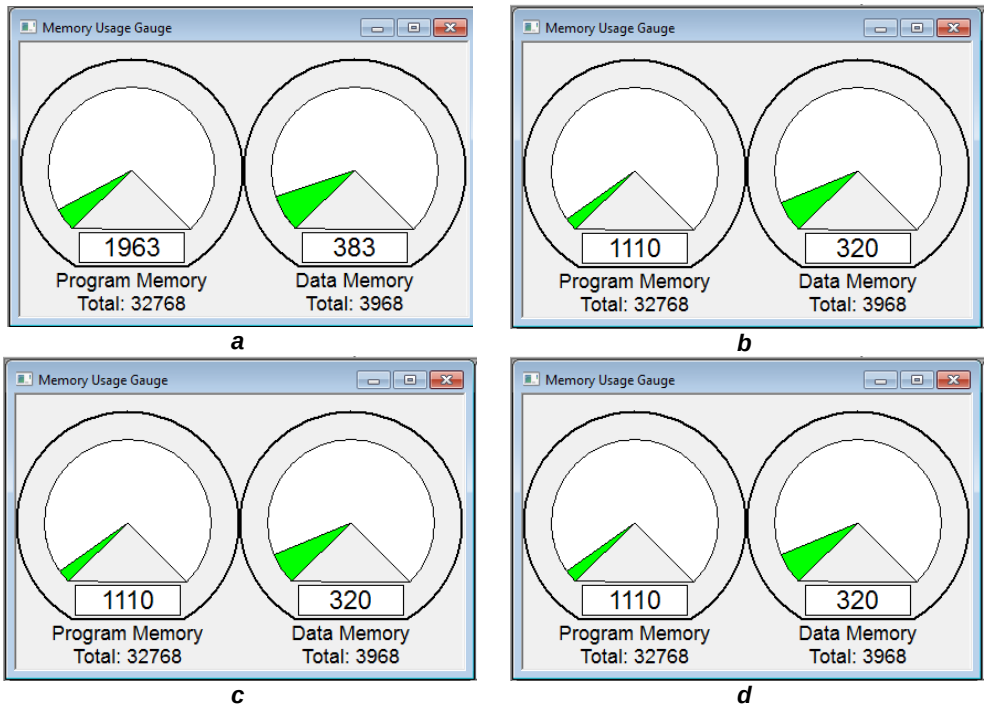


Figura 4.20 - Recursos utilizados por cada microcontrolador para implementação do sistema. a) Carro 1; b) Carro 2; c) Carro 3; e d) Carro 4.

4.2.1.3.2 Recursos utilizados e consumos com o Carro1, o Carro2 e o Carro3 implementados no Kit Spartan-III e o Carro4 num microcontrolador PIC18F4620

Mantendo os sensores e os actuadores (dos carros 1, 2 e 3) associados aos interruptores e LEDS já descritos em 4.2.1.2.1, a Tabela 4.8 e a Figura 4.21

mostram os recursos utilizados e também o consumo de energia. Os recursos utilizados pelo microcontrolador são mostrados na Figura 4.22.

Tabela 4.8 - Sumário da utilização do dispositivo utilizando a solução proposta com 3 carros na plataforma Spartan-III

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,660	3,840	43%
Number of 4 input LUTs	2,103	3,840	54%
Number of occupied Slices	1,267	1,920	65%
Number of Slices containing only related logic	1,267	1,267	100%
Number of Slices containing unrelated logic	0	1,267	0%
Total Number of 4 input LUTs	2,183	3,840	56%
Number used as logic	1,928		
Number used as a route-thru	80		
Number used as Shift registers	175		
Number of bonded IOBs	15	173	8%
IOB Flip Flops	7		
Number of BUFGMUXs	1	8	12%
Average Fanout of Non-Clock Nets	4.39		

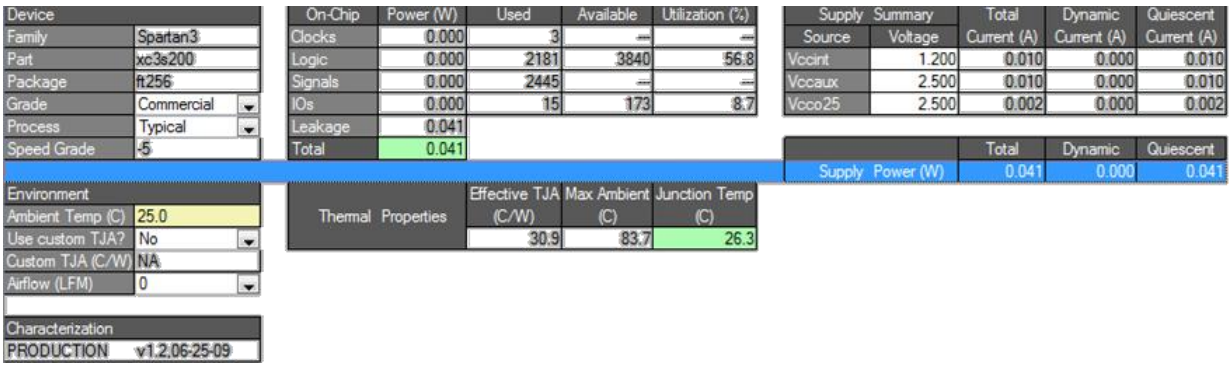


Figura 4.21 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta com 3 carros na plataforma Spartan-III

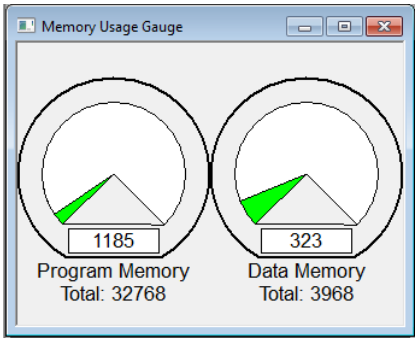


Figura 4.22 - Sumário dos recusrsos utilizados pelo PIC18F4620 utilizando a solução proposta para o carro 4

4.2.1.3.3 Recursos utilizados e consumos com os carros 1, 2 e 3 implementados no kit XUPV2C e o Carro4 mantendo-se no PIC18F4620

Neste cenário, tal como no anterior, mantiveram-se os sensores e actuadores como descrito em 4.2.1.2.2. A Tabela 4.9 e a Figura 4.23 mostram os recursos utilizados e também o consumo de energia. Os recursos utilizados pelo microcontrolador permanecem os indicados na Figura 4.22.

Tabela 4.9 - Sumário da utilização do dispositivo utilizando a solução proposta com 3 carros na plataforma XUPV2P

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,664	27,392	6%
Number of 4 input LUTs	2,045	27,392	7%
Logic Distribution			
Number of occupied Slices	1,257	13,696	9%
Number of Slices containing only related logic	1,257	1,257	100%
Number of Slices containing unrelated logic	0	1,257	0%
Total Number of 4 input LUTs	2,125	27,392	7%
Number used as logic	1,870		
Number used as a route-thru	80		
Number used as Shift registers	175		
Number of bonded IOBs			
Number of bonded	15	556	2%
Number of BUFGMUXs	1	16	6%

Name	Value	Used	Total Available	Utilization (%)
Clocks	0.00893 (W)	3	—	—
Logic	0.00117 (W)	2125	27392	7.8
Signals	0.00891 (W)	3816	—	—
IOs	0.00141 (W)	15	588	2.6
Total Quiescent Power	**0.10313 (W)**			
Total Dynamic Power	**0.02044 (W)**			
Total Power	**0.12357 (W)**			
Junction Temp	**25.0 (degrees C)**			

Figura 4.23 – Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta com 3 carros na plataforma XUPV2P

4.2.2 Exemplo II – Células de manufactura

Nesta secção, apresenta-se um exemplo da área de automação para ilustrar a aplicabilidade e utilidade das ferramentas acima referidas. O exemplo escolhido refere-se a um sistema que permite o controlo de quatro células de manufactura, composto por cinco tapetes rolantes, onde o último apenas serve como saída (Figura 4.24). Todos os outros contêm sensores que indicam a presença de objecto no início e no fim do tapete. Este último só tem um sensor no início, sinalizando que

Figura 4.24 - Sistema de manufactura sob controlo

o quarto robô acabou de tirar a peça da última célula. Estes sensores serão modelados com sinais de entrada do sistema e serão associados às transições do modelo RdP.

Considerando as RdP-IOPT como formalismo de modelação para o comportamento do controlador do sistema, obteve-se a rede apresentada na Figura 4.25. Considerando ainda que se pretende ter um controlador por cada célula (robô mais tapete) ter-se-á que decompor o modelo apresentado (na Figura 4.25) em quatro submodelos, tal como foi feito no exemplo anterior. Na Figura 4.25, os círculos encarnados representam o conjunto de corte por onde se irá dividir o sistema.

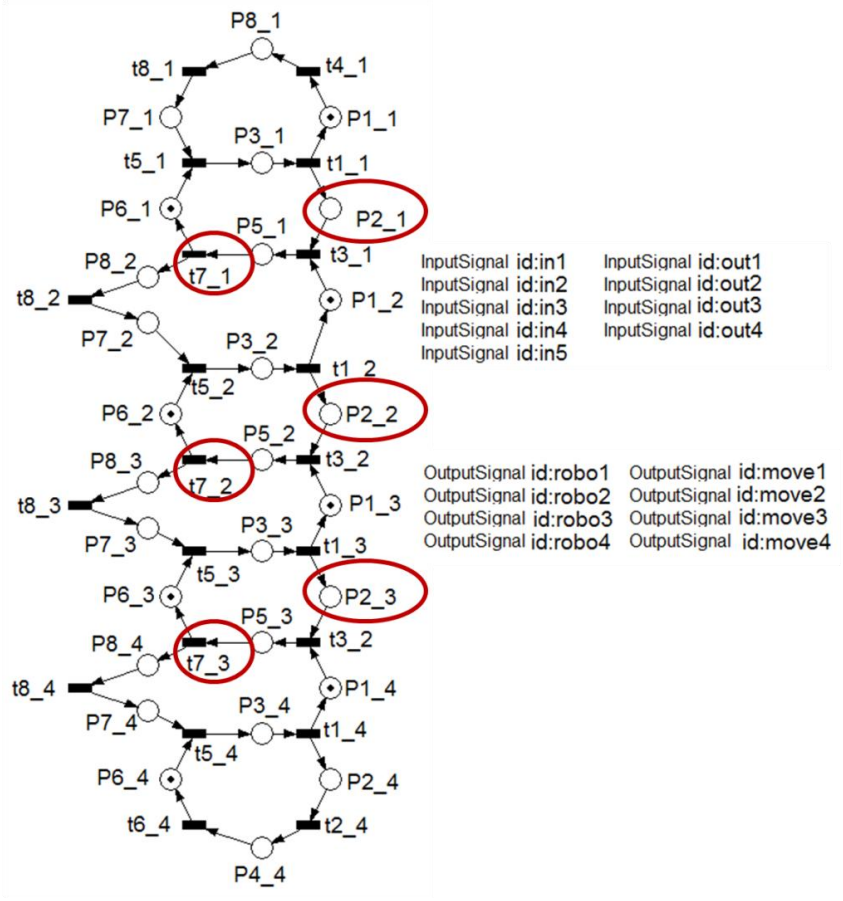


Figura 4.25 - Modelo RdP do controlador do sistema de manufactura, com o conjunto de corte assinalado

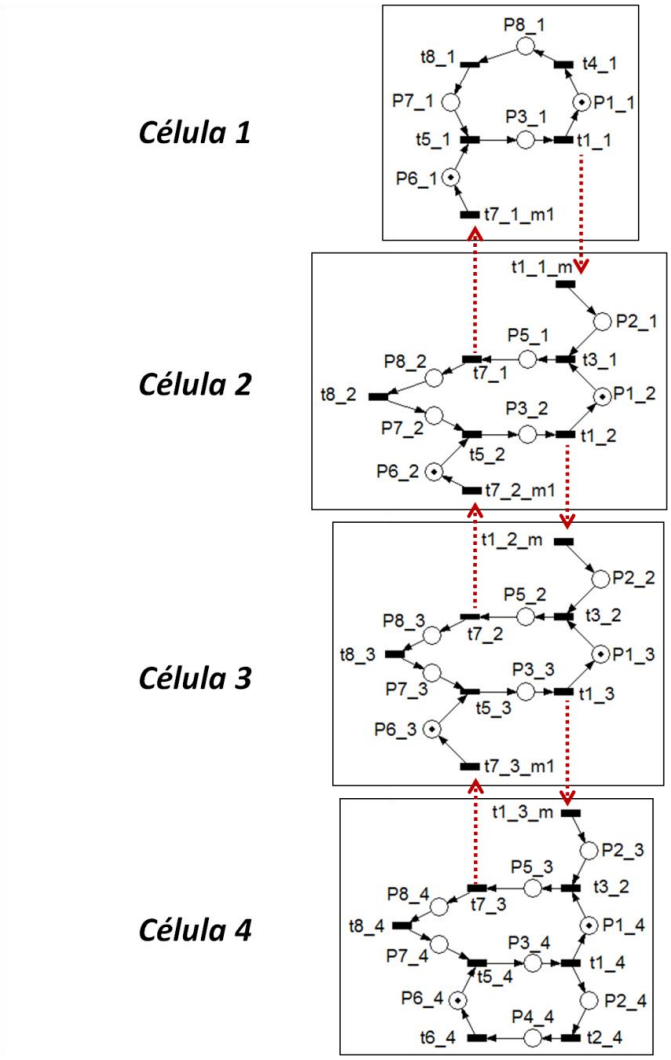


Figura 4.26 - Modelos dos controladores das várias células.

Tal como já foi referido, em 4.2.1, a ferramenta SPLIT, para além de obter os modelos para cada célula, gera os eventos de comunicação entre modelos. Na Figura 4.26 estão representados os modelos que serão implementados por cada controlador.

Tabela 4.10 - Sinais associados a cada transição / lugar (Exemplo II)

Sinais de entrada (sensores)		*Sinais de saída· (actuadores)*	
Transição	**Sinal**	**Lugar**	**Sinal**
t4_1	in1	P8_1	move1
t8_1	out1	P5_1	robo1
t7_1	in2	P8_2	move2
t8_2	out2	P5_2	robo2
t7_2	in3	P8_3	move3
t8_3	out3	P5_3	robo3
t7_3	in4	P8_4	move4
t8_4	out4	P4_4	robo4
t6_4	in5		

Para não sobrecarregar a figura, não estão visíveis os sinais de actuação dos robôs e motores, nem os sinais dos sensores in# e out#. A Tabela 4.10 mostra onde está cada sinal do modelo (Figura 4.25 e Figura 4.26).

4.2.2.1 Ligação entre componentes numa só plataforma sem recorrer à solução proposta

Mais uma vez, utilizou-se a ferramenta realizada em [28], para interligar os eventos dos quatro módulos e obter os recursos utilizados pelo sistema (descritos na Tabela 4.11) antes de aplicar a solução proposta. O consumo de potência previsto é descrito na Figura 4.27.

Tabela 4.11 - Sumário da utilização do dispositivo utilizando o cenário de aplicação depois de aplicar a ferramenta SPLIT, ligações ponto-a-ponto, ao exemplo II

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	55	3,840	1%
Number of 4 input LUTs	46	3,840	1%
Number of occupied Slices	52	1,920	2%
Number of Slices containing only related logic	52	52	100%
Number of Slices containing unrelated logic	0	52	0%
Total Number of 4 input LUTs	46	3,840	1%
Number of bonded IOBs	19	173	10%
Number of BUFGMUXs	1	8	12%
Average Fanout of Non-Clock Nets	2.69		

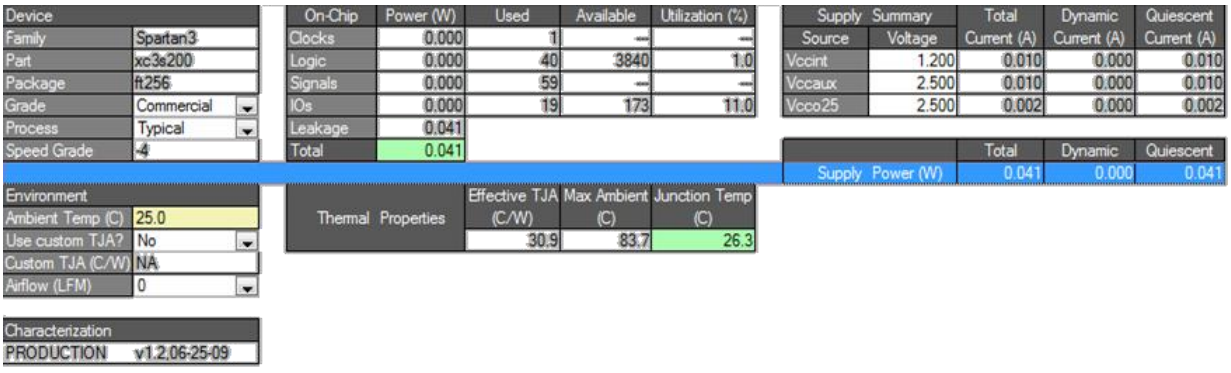


Figura 4.27 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando o cenário de aplicação depois de aplicar a ferramenta SPLIT, ligações ponto-aponto, ao exemplo II

O sistema foi implementado no Kit Spartan-III, utilizando os oito interruptores e um botão de pressão para simular os sensores, e os LEDs para simular os actuadores dos tapetes e robôs.

Também neste exemplo, o resultado obtido foi o esperado, tendo o sistema distribuído o mesmo comportamento que o sistema do modelo inicial.

4.2.2.2 Ligação entre componentes numa só plataforma

Recorrendo à aplicação construída, onde foram implementados os algoritmos e regras descritos em 3.3.1, definir-se-ão os endereços e a topologia da rede, tal como já foi feito para o exemplo anterior. O autor, após ter inserido o número de nós, fornece o número de mensagens trocadas entre nós (Figura 4.28). As velocidades entre nós são as mesmas que as do exemplo anterior (Figura 4.10).

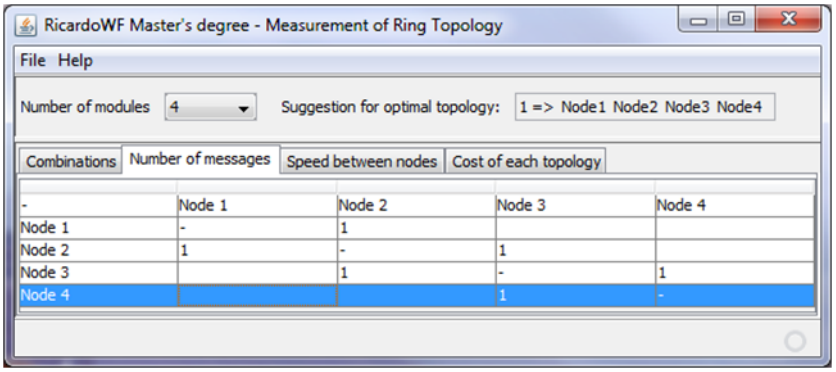


Figura 4.28 - Inserção do número de mensagens trocadas entre nós (Exemplo II)

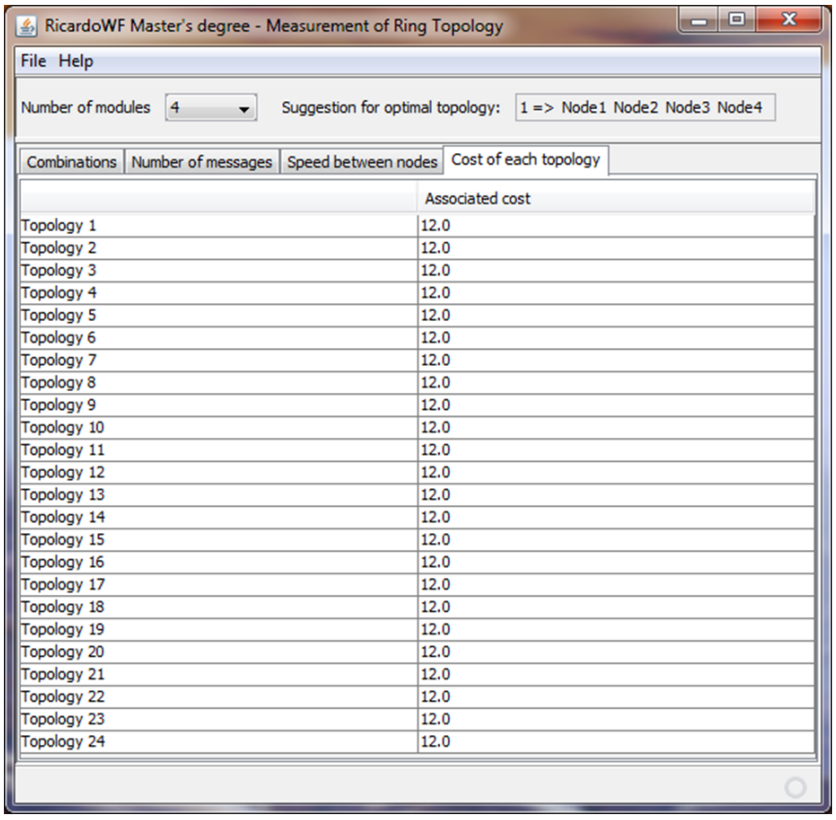


Figura 4.29 - Custo associado a cada topologia (Exemplo II)

A escolha recai na topologia *Nó1 – Nó2 – Nó3 – Nó4*, com um custo associado de 12 (como se verifica na Figura 4.29). Apesar de neste exemplo, ao contrário do apresentado anteriormente, se estar com dois módulos que enviam e recebem uma mensagem, e outros dois que enviam e recebem duas, o valor mantém-se constante independentemente da topologia, dado estar-se numa topologia em anel e não em duplo anel. Qualquer que seja a maneira de interligar os nós o custo é o mesmo. Tal não aconteceria se se tivesse uma topologia em duplo anel. Tendo em conta este resultado, é atribuído o endereço #00 à célula 1, #01 à célula 2, #10 à célula 3 e #11 à célula 4. Para se poderem compor as mensagens que circularão na rede (Figura 4.31), os sinais, previamente obtidos dos ficheiros PNML, são codificados, como mostra a Tabela 4.12.

Tabela 4.12 - Codificação dos eventos de entrada de cada célula

Célula1 - #00

Evento	Código
in1	-
out1	-
inevent2416	0

Célula2 - #01

Evento	Código
in2	-
out2	-
inevent2380	0
inevent2774	1

Célula3 - #10

Evento	Código
in3	-
out3	-
inevent2738	0
inevent4415	1

Célula4 - #11

Evento	Código
in4	-
out4	-
in5	-
inevent4451	0

Tal como no exemplo anterior, os módulos hardware foram implementados numa só plataforma (kit Spartan-III e kit XUPV2P), como mostra a Figura 4.30.

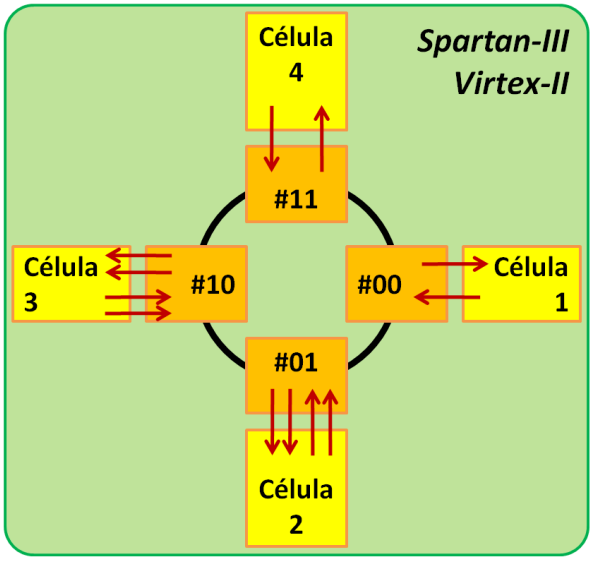


Figura 4.30 - Implementação hardware numa só plataforma (Exemplo II)

Foram consideradas as mesmas velocidades de relógio, ou seja, a implementação no kit Spartan-III terá a: célula 1 – 50 MHz; célula 2 – 32,7 MHz; célula 3 – 24 MHz; e célula 4 – 10 MHz; e a implementação no kit XUPV2P terá a: célula 1 – 32 MHz; célula 2 – 32 MHz desfasado; célula 3 – 100 MHz; e célula 4 –

100 MHz desfasado. A velocidade de relógio dos nós também se mantém, 50 MHz para a primeira plataforma, e 100 MHz para a segunda, conservando os *baund_count_max* anteriormente calculados (1 e 2, respectivamente, consoante a plataforma).

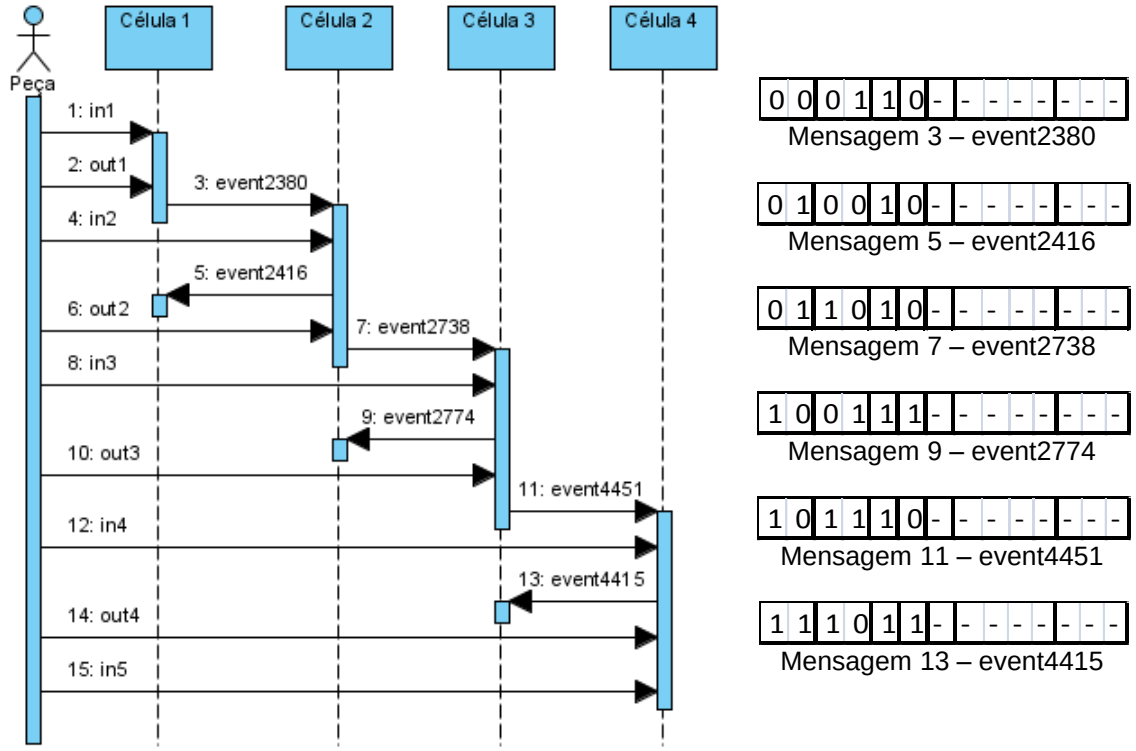


Figura 4.31 - Mensagens a circular na rede (Exemplo II)

A Tabela 4.13 mostra o valor obtido dos sinais genéricos a ser preenchido nos *templates*.

Tabela 4.13 - Atribuição dos valores dos sinais genéricos (Exemplo II)

Sinal genérico	**Célula 1**	**Célula 2**	**Célula 3**	**Célula 4**
MaxLenght	2			
DimMsg	14			
Dim_Node	2			
Dim_IDsignal	1			
Dim_Signal	1			
DIm_CRC	3			
CurrentNode	00	01	10	11
DestinationNode1	01	00	01	10
Node1SinalID	0	0	1	1
DestinationNode2	-	10	11	-
Node2SinalID	-	0	0	-
Num_signalsSend	1	1	1	1
Buffer_Lenght	4			
Vector_width	13			
Num_signalsReceiv	1	2	2	1
Num_sinais	1	2	2	1

Neste exemplo, e ao contrário do outro, foi considerado o pior cenário para a dimensão dos *buffers*, ou seja, quatro posições de memória (uma para cada evento, outra para o caso de se estar a receber uma mensagem nesse momento, e outra de reserva).

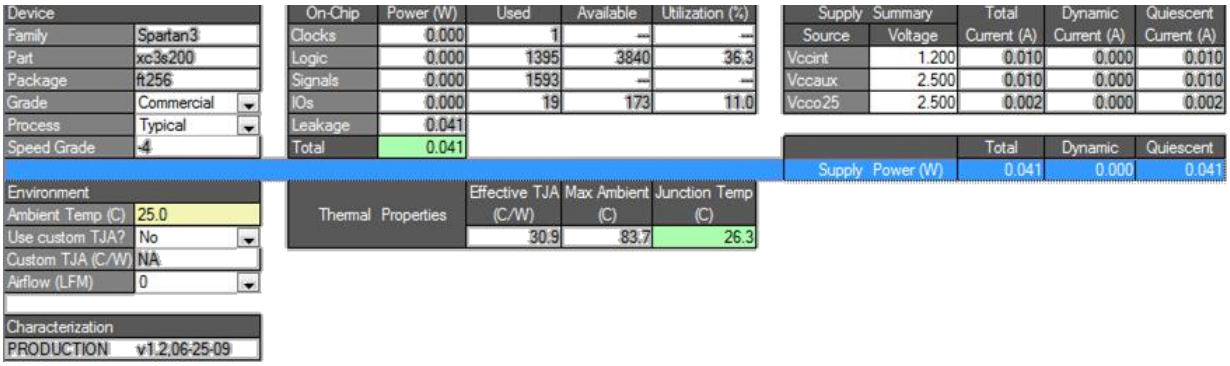
Tabela 4.14 - Instantes de tempo de cada sinal do exemplo da Figura 4.31, obtidos através do ModelSim

Sinal	*Instante de tempo [ms]*
IN1	0,000045
Tapete1	0,000055
OUT1	1,000045
outevent2380	1,500045
inevent2380	1,503385
Robo1	1,503385
IN2	2,000045
outevent2416	2,000045
Tapete2	2,000055
intevent2416	2,009915
OUT2	3,000045
outevent2738	3,500045
inevent2738	3,503375
Robo2	3,503385
IN3	4,000045
outevent2774	4,000045
Tapete3	4,000055
inevent2774	4,009915
OUT3	5,000045
outevent4451	5,500045
inevent4451	5,503375
Robo3	5,503385
IN4	6,000045
outevent4415	6,000045
Tapete4	6,000055
inevent4451	6,009915
OUT4	7,000045
Robo4	7,500055
IN5	8,000045

Para se saber o tempo de propagação entre os eventos de comunicação, gerados pela ferramenta SPLIT, utilizando a solução proposta, foram realizadas as medições ilustradas na Tabela 4.14.

4.2.2.2.1 Recursos utilizados e consumos na plataforma Spartan-III

Recorrendo à mesma configuração que a utilizada na implementação anterior, os recursos usados e o consumo de energia são mostrados na Tabela 4.15 e Figura 4.32, respectivamente.



Device	
Family	Spartan3
Part	xc3s200
Package	ft256
Grade	Commercial
Process	Typical
Speed Grade	-4

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.000	1	--	--
Logic	0.000	1395	3840	36
Signals	0.000	1593	--	--
IOs	0.000	19	173	11
Leakage	0.041			
Total	0.041			

Supply Summary		Total	Dynamic	Quiescent
Source	Voltage	Current (A)	Current (A)	Current (A)
Vccint	1.200	0.010	0.000	0.010
Vccaux	2.500	0.010	0.000	0.010
Vcco25	2.500	0.002	0.000	0.002

	Total	Dynamic	Quiescent
Supply Power (W)	0.041	0.000	0.041

Environment	
Ambient Temp (C)	25.0
Use custom TJA?	No
Custom TJA (C/W)	NA
Airflow (LFM)	0

Characterization	
PRODUCTION	v1.2,06-25-09

Thermal Properties	Effective TJA (C/W)	Max Ambient (C)	Junction Temp (C)
	30.9	83.7	26.3

Figura 4.32 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta só na plataforma Spartan-III (Exemplo II).

Tabela 4.15 - Sumário da utilização do dispositivo utilizando a solução proposta só na plataforma Spartan-III (Exemplo II).

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,060	3,840	27%
Number of 4 input LUTs	1,399	3,840	36%
Number of occupied Slices	843	1,920	43%
Number of Slices containing only related logic	843	843	100%
Number of Slices containing unrelated logic	0	843	0%
Total Number of 4 input LUTs	1,399	3,840	36%
Number used as logic	1,259		
Number used as Shift registers	140		
Number of bonded IOBs	19	173	10%
Number of BUFGMUXs	1	8	12%
Average Fanout of Non-Clock Nets	4.27		

4.2.2.2.2 Recursos utilizados e consumos na plataforma XUPV2P

Nesta plataforma, e como só existem disponíveis quatro interruptores e cinco botões de pressão, foram utilizados os interruptores do IMLAB1 para simular os sensores e os botões de sincronismo. Pela mesma razão, não foram utilizados os LEDs da plataforma e sim os do IMLAB1.

A Tabela 4.16 e a Figura 4.33 mostram os recursos utilizados e também o consumo de energia.

Tabela 4.16 -Sumário da utilização do dispositivo utilizando a solução proposta só na plataforma XUPV2P (Exemplo II).

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,045	27,392	3%
Number of 4 input LUTs	1,463	27,392	5%
Logic Distribution			
Number of occupied Slices	869	13,696	6%
Number of Slices containing only related logic	869	869	100%
Number of Slices containing unrelated logic	0	869	0%
Total Number of 4 input LUTs	1,463	27,392	5%
Number used as logic	1,323		
Number used as Shift registers	140		
Number of bonded IOBs			
Number of bonded	19	556	3%
Number of BUFGMUXs	1	16	6%

Name	Value	Used	Total Available	Utilization (%)
Clocks	0.00842 (W)	1	—	—
Logic	0.00067 (W)	1459	27392	5.3
Signals	0.00547 (W)	2499	—	—
IOs	0.00757 (W)	19	588	3.2
Total Quiescent Power	**0.10313 (W)**			
Total Dynamic Power	**0.02213 (W)**			
Total Power	**0.12526 (W)**			
Junction Temp	**25.0 (degrees C)**			

Figura 4.33 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta só na plataforma XUPV2P (Exemplo II)

4.2.2.3 Ligação entre componentes em plataformas heterogéneas

Tal como anteriormente, serão consideradas três hipóteses de interligação dos módulos: (1) Célula 1, Célula 2 e Célula 4 implementadas no Kit Spartan-III e a Célula 3 num microcontrolador PIC18F4620; (2) Em vez do kit Spartan-III, as células um, dois e quatro são implementadas no kit XUPV2C e a célula 4 mantém-se no PIC18F4620; e (3) ter cada uma das células implementadas num PIC18F4620. Também neste caso, para a velocidade de transmissão entre os módulos dos microcontroladores, foi considerada uma taxa de transmissão de 9600 bps.

À semelhança do que aconteceu para o exemplo anterior, foi sugerida a mesma topologia, quer para a terceira hipótese, quer para as duas primeiras hipóteses. Assim, as codificações dos sinais / eventos mantêm-se as da Tabela 4.12, conservando as mensagens mostradas na Figura 4.31.

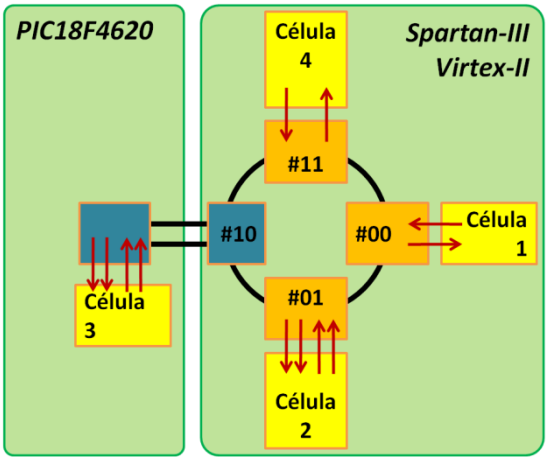


Figura 4.34 - Implementação em plataformas heterogéneas (entre kits Xilinx e microcontrolador - Exemplo II)

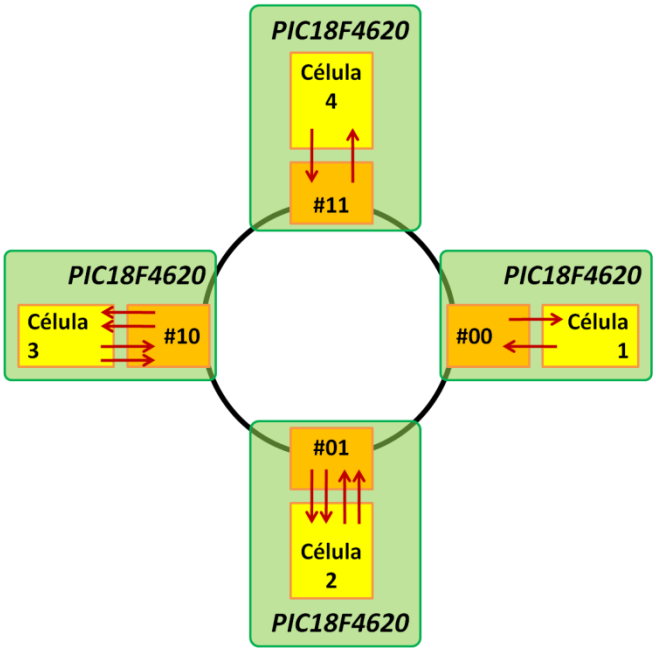


Figura 4.35 - Implementação em plataformas heterogéneas (entre quatro microcontroladores - Exemplo II)

A Figura 4.34 e a Figura 4.35 mostram como é feita a interligação entre as plataformas para as três hipóteses consideradas. Basicamente, são semelhantes às da Figura 4.18 e da Figura 4.19. A diferença será que em vez dos controladores dos carros ter-se-ão os controladores de cada uma das células, com os respectivos eventos ligados aos nós de rede, que no caso da Figura 4.18 (interligação entre kits Xilinx e o microcontrolador) o controlador que ficou no PIC18F4620 passa a ser o da célula 3.

Para a descrição de hardware mantêm-se os sinais genéricos, apresentados na Tabela 4.13, com excepção dos sinais *baund_count_max* de envio do nó da

célula 2 e de recepção do nó da célula 4. Com uma taxa de 9600 bps, serão dados por: $\mathrm{EN\_16\_X\_BAUD} = 16 \times 9600\ \mathrm{Hz} = 153{,}6\ KHz$, o que implica ter no kit Spartan-III um

$$baud\_count\_max = \frac{50\ MHz\ (velocidade\ de\ relógio)}{153{,}6\ KHz\ (EN\_16\_X\_BAUD)} = 326.$$

e no kit XUPV2P um

$$baud\_count\_max = \frac{100\ MHz\ (velocidade\ de\ relógio)}{153{,}6\ KHz\ (EN\_16\_X\_BAUD)} = 651.$$

Em software foi configurada a UART do microcontrolador da mesma forma que no exemplo anterior, e dado que as características se mantêm, ter-se-á um

$$spbrg = \frac{\frac{4{,}9152 \times 10^6\ \ (velocidade\ do\ oscilaor)}{9600\ \ (ritmo\ desejado)}}{16} - 1 = 31$$

4.2.2.3.1 Recursos utilizados em cada uma das células implementadas num PIC18F4620

Mantendo o mesmo ambiente de implementação, e voltando a associar os interruptores (sensores IN e OUT) com o registo PORTB do microcontrolador e o registo PORTD para os actuadores dos tapetes e robots, a Figura 4.36 mostra os recursos utilizados por cada PIC18F4620 para a execução do sistema.

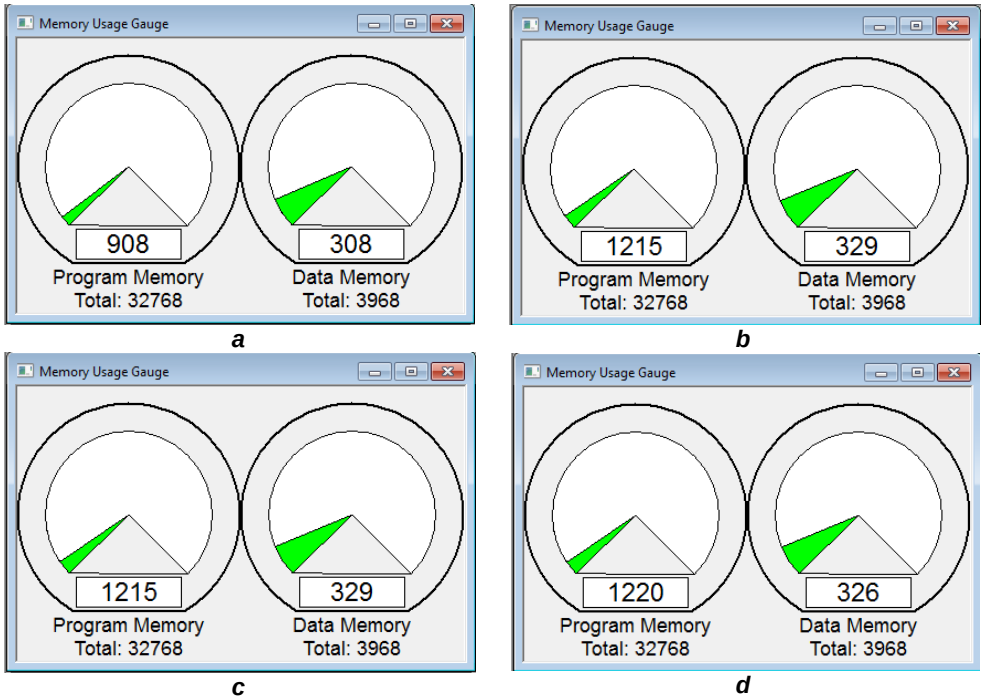


Figura 4.36 - Recursos utilizados por cada microcontrolador para implementação do sistema. a) Célula 1; b) Célula 2; c) Célula 3; e d) Célula 4.

4.2.2.3.2 Recursos utilizados e consumos com a Célula 1, a Célula 2 e a Célula 4 implementadas no Kit Spartan-III e a Célula 3 num microcontrolador PIC18F4620

Conservando sensores e actuadores associados aos interruptores e LEDS já descritos em 4.2.2.2, a Tabela 4.17 e a Figura 4.37 mostram os recursos utilizados e também o consumo de energia. Os recursos utilizados pelo microcontrolador são mostrados na Figura 4.38.

Tabela 4.17 - Sumário da utilização do dispositivo utilizando a solução proposta com 4 células na plataforma Spartan-III

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,255	3,840	32%
Number of 4 input LUTs	1,728	3,840	45%
Number of occupied Slices	1,046	1,920	54%
Number of Slices containing only related logic	1,046	1,046	100%
Number of Slices containing unrelated logic	0	1,046	0%
Total Number of 4 input LUTs	1,744	3,840	45%
Number used as logic	1,553		
Number used as a route-thru	16		
Number used as Shift registers	175		
Number of bonded IOBs	17	173	9%
Number of BUFGMUXs	1	8	12%
Average Fanout of Non-Clock Nets	4.33		

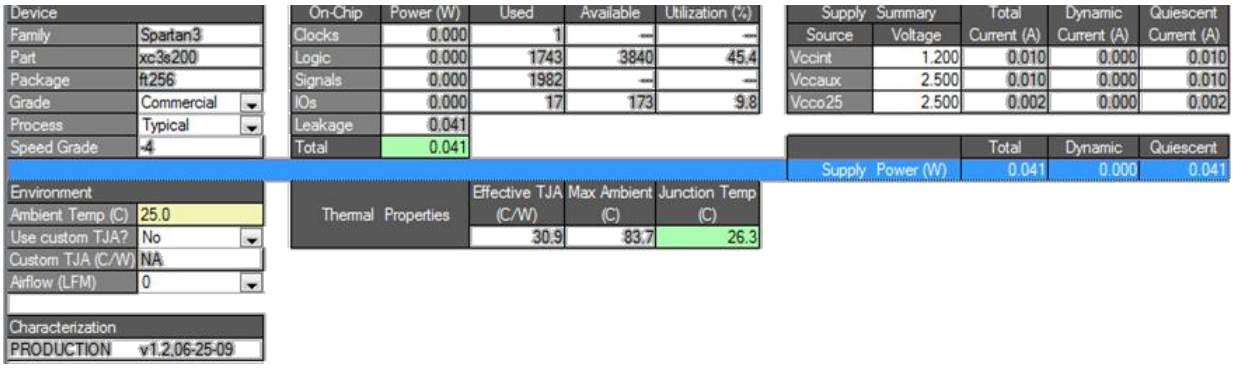


Figura 4.37 - Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta com 3 células na plataforma Spartan-III

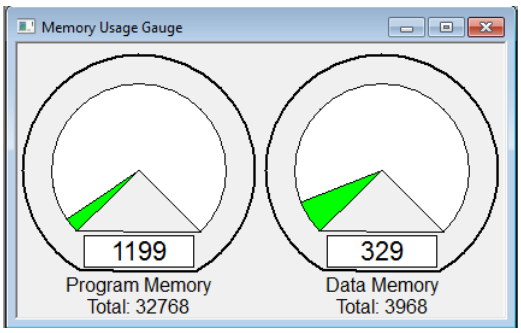


Figura 4.38 - Sumário dos recursos utilizados pelo PIC18F4620 utilizando a solução proposta para a célula 3

4.2.2.3.3 Recursos utilizados e consumos com as células 1, 2 e 4 implementadas no kit XUPV2C e a Célula 3 mantendo-se no PIC18F4620

Mantendo-se os sensores e actuadores, a Tabela 4.18 e a Figura 4.39 mostram os recursos utilizados e o consumo de energia. Os recursos usados pelo microcontrolador mantêm-se os indicados na Figura 4.38.

Tabela 4.18 - Sumário da utilização do dispositivo utilizando a solução proposta com 3 células na plataforma XUPV2P

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Flip Flops	1,229	27,392	4%
Number of 4 input LUTs	1,767	27,392	6%
Logic Distribution			
Number of occupied Slices	1,058	13,696	7%
Number of Slices containing only related logic	1,058	1,058	100%
Number of Slices containing unrelated logic	0	1,058	0%
Total Number of 4 input LUTs	1,783	27,392	6%
Number used as logic	1,592		
Number used as a route-thru	16		
Number used as Shift registers	175		
Number of bonded IOBs			
Number of bonded	17	556	3%
Number of BUFGMUXs	1	16	6%

Name	Value	Used	Total Available	Utilization (%)
Clocks	0.00886 (W)	1	—	—
Logic	0.00048 (W)	1782	27392	6.5
Signals	0.00445 (W)	3023	—	—
IOs	0.00580 (W)	17	588	2.9
Total Quiescent Power	**0.10313 (W)**			
Total Dynamic Power	**0.01959 (W)**			
Total Power	**0.12271 (W)**			
Junction Temp	**25.0 (degrees C)**			

Figura 4.39 -Tabelas, retiradas de Xilinx XPower Analyzer, com a previsão da potência consumida utilizando a solução proposta com 3 células na plataforma XUPV2P

5 Conclusões e Perspectivas Futuras

As regras e considerações apresentadas neste trabalho permitem a interligação de diversos componentes inicialmente expressos por modelos de redes de Petri IOPT, através de uma rede de comunicação baseada num anel satisfazendo o protocolo de comunicação série RS-232 (embora operando com ritmos de transmissão mais elevados). Assim, e apesar de ser uma solução de suporte à comunicação potencialmente mais lenta que outras soluções NoC, é uma solução muito flexível que permitirá a sua utilização num número elevado de plataformas em que se pretendam instalar os componentes associados aos submodelos concorrentes, bem como a interligação a soluções proprietárias que satisfaçam o protocolo RS-232.

Existem actualmente no mercado inúmeros adaptadores do protocolo RS-232 para outros protocolos, nomeadamente Ethernet, Bluetooth, USB, entre outros. Para o desenvolvimento deste trabalho foi realizada uma aplicação em Visual Studio .NET 2008, da Microsoft, cuja interface é apresentada na Figura 5.1, para fazer debug enviando / recebendo mensagens do nó, verificando-se qual o resultado. Como nem todos os computadores para realizar este projecto tinham porta série (RS-232), usou-se um adaptador USB | RS-232 onde, apesar de em nenhum dos exemplos mostrados atrás terem sido implementados num computador, os resultados foram os

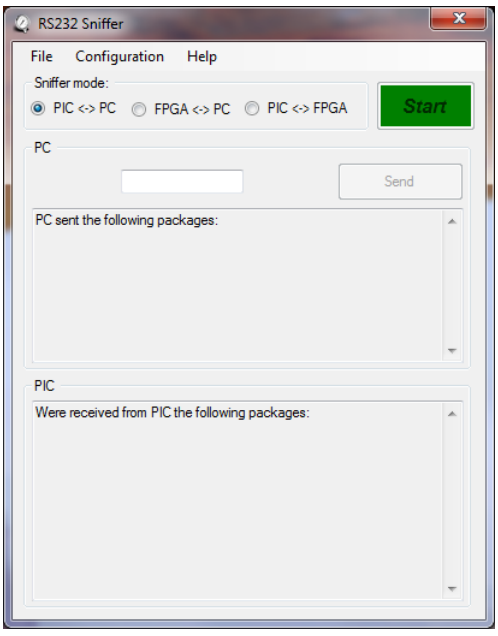


Figura 5.1 - Aplicação RS232 Sniffer desenvolvida para auxiliar na monitorização das mensagens a circular na rede.

esperados. A escolha pelo protocolo RS-232 aumenta assim a versatilidade da solução.

A metodologia proposta beneficia das vantagens associadas às metodologias de desenvolvimento baseadas em modelos, permitindo um tempo de desenvolvimento potencialmente mais curto e sendo menos sensível a erros de codificação, uma vez que se utilizam ferramentas de geração automática de código. Contribui, também, para a utilização das ferramentas geradas no projecto FORDESIGN e para o reforço da utilização de redes de Petri no desenvolvimento de sistemas embutidos com base em plataformas reconfiguráveis, mantendo-se a capacidade de interligação a sistemas externos (heterogéneos).

O conjunto de regras de construção da rede foi definido (nomeadamente o método para atribuição do endereço a cada um dos nós) e a solução validada através dos exemplos apresentados, cuja Tabela 5.1 sumariza os resultados obtidos.

Tabela 5.1 - Sumários dos resultados obtidos com os dois exemplos implementados

		Exemplo I 4 Carros sincronizados				**Exemplo II** 4 Células de manufactura			
		Flip Flops	***LUTs***	***Slices***	***IOBs***	***Flip Flops***	***LUTs***	***Slices***	***IOBs***
Uma plataforma sem a solução proposta	***Spartan3***	28 (1%)	41 (1%)	28 (1%)	16 (9%)	55 (1%)	46 (1%)	52 (2%)	19 (10%)
	Virtex-II	34 (1%)	41 (1%)	34 (1%)	16 (2%)	37 (1%)	40 (1%)	30 (1%)	19 (3%)
Uma plataforma	***Spartan3***	1622 (42%)	2079 (54%)	1243 (64%)	16 (9%)	1060 (27%)	1399 (36%)	842 (43%)	19 (10%)
	Virtex-II	1624 (5%)	2006 (7%)	1214 (8%)	16 (2%)	1045 (3%)	1463 (5%)	869 (6%)	19 (3%)
Plataformas heterogéneas	***Spartan3***	1660 (43%)	2103 (54%)	1267 (65%)	15 (8%)	1255 (32%)	1728 (45%)	1046 (54%)	17 (9%)
	Virtex-II	1664 (6%)	2045 (7%)	1257 (9%)	15 (2%)	1229 (4%)	1767 (6%)	1058 (7%)	17 (3%)
	PIC	1185 (4%)		323 (8%)		1199 (4%)		329 (8%)	
		Memória de programa		***Memória de dados***		***Memória de programa***		***Memória de dados***	

O trabalho desenvolvido contribuiu para uma comunicação apresentada nas jornadas REC2010 com uma visão geral da arquitectura deste trabalho e o respectivo enquadramento [46]. Foi também submetida uma comunicação a uma conferência internacional de reconhecida qualidade, dando ênfase aos resultados obtidos, não estando ainda concluído o seu processo de revisão.

O passo seguinte é o de desenvolver uma ferramenta que permita automatizar o processo, aplicando as regras descritas e os *templates* criados de forma a gerar automaticamente o código necessário para cada plataforma.

O segundo passo poderá ser o de implementar uma topologia em duplo anel aumentando assim a fiabilidade e a tolerância a falhas. Para tal, cada nó passaria a ter uma tabela de encaminhamento, optando sempre pelo caminho mais rápido. Sempre que exista uma avaria ter-se-á sempre como opção o funcionamento em anel. Esta hipótese aumentará os recursos utilizados, mas simultaneamente acrescentará a fiabilidade.

A comparação com outros tipos de Networks-on-Chip e a identificação de regras que permitam a construção de uma topologia híbrida serão também objecto de trabalho futuro sem nunca descurar a ligação com sistemas externos, nomeadamente computadores de utilização geral e microcontroladores de baixo custo. Dados os resultados da Tabela 5.1, faz sentido verificar os custos de outras soluções, nomeadamente soluções do tipo GALS, de forma a, mantendo os objectivos do presente trabalho, tentar diminuir os custos, sempre de maneira transparente ao utilizador que modela o sistema.

Referências

[1] Microchip, "PIC18F2525/2620/4525/4620 Data Sheet," DS39626B ed: Microchip Technology Inc., 2004, p. 390.

[2] ATMEL, "16-word by 8-bit FIFO, Application Note," 0473C-09/99 ed: ATMEL, 1999, p. 5.

[3] A. V. Levitin, *Introduction to the Design and Analysis of Algorithms*, 2 edition ed.: Addison Wesley, 2003.

[4] "FORDESIGN – Formal Methods for Embedded Systems Co-Design.," in *R&D project POSC/EIA/61364/2004 sponsored by FCT - Fundação para a Ciência e a Tecnologia.*: www.uninova.pt/fordesign, 2007.

[5] P. N. Victor, H. T. Nagle, D. C. Bill, and J. D. Irwin, *Digital logic circuit analysis and design*: Prentice-Hall, Inc., 1995.

[6] Xilinx, "Xilinx University Program Virtex-II Pro Development System - Hardware Reference Manual," UG069(v1.2) ed: Xilinx, Inc., 2009, p. 142.

[7] L. Gomes and J. M. Fernandes, *Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation*: IGI Global; Information Science Reference, 2009.

[8] D. Gajski, S. Abdi, A. Gerstlauer, and G. Sshirner, *Embedded System Design Modeling, Synthesis and Verification*: Springer, 2009.

[9] L. Gomes, J. P. Barros, A. Costa, and R. Nunes, "The Input-Output Place-Transition Petri Net Class and Associated Tools," in *Industrial Informatics, 2007 5th IEEE International Conference on*, 2007, pp. 509-514.

[10] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The Petri Net Markup Language: Concepts, Technology, and Tools," in *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003. Proceedings*, 2003, pp. 1023-1024.

[11] R. Wolfgang, *Petri nets: an introduction*: Springer-Verlag New York, Inc., 1985.

[12] R. Pais, J. P. Barros, and L. Gomes, "A tool for tailored code generation from Petri net models," in *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, 2005, pp. 8 pp.-864.

[13] A. Costa and L. Gomes, "Petri net Splitting Operation within Embedded Systems Co-design," in *Industrial Informatics, 2007 5th IEEE International Conference on*, 2007, pp. 503-508.

[14] J. Nurmi, "Network-on-Chip: A New Paradigm for System-on-Chip Design," in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, 2005, pp. 2-6.

[15] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," *Circuits and Systems Magazine, IEEE,* vol. 4, pp. 18-31, 2004.

[16] D. Sigüenza-Tortosa, T. Ahonen, and J. Nurmi, "Issues in the development of a practical NoC: the Proteo concept," *Integration, the VLSI Journal,* vol. 38, pp. 95-105, 2004.

[17] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, "The Nostrum backbone-a communication protocol stack for Networks on Chip," in *VLSI Design, 2004. Proceedings. 17th International Conference on*, 2004, pp. 693-696.

[18] D. Wiklund and L. Dake, "SoCBUS: switched network on chip for hard real time embedded systems," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003, p. 8 pp.

[19] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, 2000, pp. 250-256.

[20] H. Kariniemi and J. Nurmi, "Reusable XGFT interconnect IP for network-on-chip implementations," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*, 2004, pp. 95-102.

[21] W. Xin and J. Nurmi, "An On-Chip CDMA Communication Network," in *System-on-Chip, 2005. Proceedings. 2005 International Symposium on*, 2005, pp. 155-160.

[22] K. Goossens, J. Dielissen, and A. Radulescu, "AEthereal network on chip: concepts, architectures, and implementations," *Design & Test of Computers, IEEE,* vol. 22, pp. 414-421, 2005.

[23] S. Erno, K. Ari, and D. H. Timo, "On network-on-chip comparison," in *Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*: IEEE Computer Society, 2007.

[24] B. Tobias and M. Shankar, "A survey of research and practices of Network-on-chip," *ACM Comput. Surv.,* vol. 38, p. 1, 2006.

[25] F. Leens, "An introduction to I²C and SPI protocols," *Instrumentation & Measurement Magazine, IEEE,* vol. 12, pp. 8-13, 2009.

[26] W. P. Byron, *RS-232 simplified: everything you need to know about connecting, interfacing, and troubleshooting peripheral devices*: Prentice-Hall, Inc., 1987.

[27] A. J. Martin and M. Nystrom, "Asynchronous Techniques for System-on-Chip Design," *Proceedings of the IEEE,* vol. 94, pp. 1089-1120, 2006.

[28] J. Oliveira, "Configurador de plataformas específicas em Co-design de Sistemas Embutidos," in *Departamento de Engenharia Electrotécnica*. vol. Mestre em Engenharia Electrótecnica e de Computadores Lisboa: Universidade Nova de Lisboa, 2009, p. 175.

[29] A. Costa and L. Gomes, "Petri net partitioning using net splitting operation," in *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, 2009, pp. 204-209.

[30] X. Jiang, "Design, modeling, and analysis of networks-on-chip for systems-on-chip," Princeton University, 2007, p. 180.

[31] S. Wallace, M. K. Senehi, E. Barkmeyer, S. Ray, and E. K. Wallace, "Control Entity Interface Specification," in *Manufacturing Systems Integration*, NISTIR 5272 ed U.S. Department of Commerce; Technology Administration: National Institute of Standards and Technology;, 1993, p. 118.

[32] toExcel, *Extensible Markup Language (Xml) 1.0 Specifications: From the W3c Recommendations*: iUniverse, Incorporated, 2000.

[33] M. E. Graça Martins and A. G. Cerveira, *Introdução às Probabilidades e à Estatística*: Universidade Aberta, 1999.

[34] J. Landin, *An Introduction to Algebraic Structures*: Dover Publications, 1969.

[35] E. W. Dijkstra, *A Discipline of Programming*: Prentice Hall, Inc., 1976.

[36] A. Engel, *Exploring Mathematics with your Computer*, Pap/Dis edition ed.: The Mathematical Association of America, 1997.

[37] R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3 edition ed.: Addison-Wesley Professional, 2001.

[38] Xilinx, "Xilinx Synthesis Technology (XST) User Guide," Xilinx, Inc., 2002, p. 450.

[39] K. Chapman, "Application Note: Virtex, Virtex-E, and Spartan-II Families - 200 MHz UART with Internal 16-Byte Buffer," XAPP223 (v1.2) ed: Xilinx, Inc., 2008, p. 12.

[40] T. Paul, G. Mark, and L. Guy, "A Survey and Taxonomy of GALS Design Styles," *IEEE Des. Test,* vol. 24, pp. 418-428, 2007.

[41] Xilinx, "Spartan-3 FPGA Starter Kit Board User Guide," UG130(v1.2) ed: Xilinx, 2008, p. 64.

[42] Xilinx, "Spartan-3 FPGA Family Data Sheet," DS099 ed: Xilinx Inc., 2009, p. 217.

[43] Xilinx, "Virtex-II Pro and Virtex-II Pro X Plataform FPGAs: Complete Data Sheet," DS083 ed: Xilinx Inc., 2007, p. 430.

[44] M. Silva, "Las redes de Petri en la automÃ¡tica y la informatica / Manuel Silva," Madrid : Editorial AC, 1985.

[45] Microchip, "MPLAB C18 C Compiler Libraries," DS51297F ed: Microchip Tecnhonoly Inc., 2005, p. 184.

[46] R. Ferreira, A. Costa, and L. Gomes, "Interligação Intra- e Inter-circuito de Componentes Especificados com Redes de Petri," in *VI Jornadas sobre Sistemas Reconfiguráveis - REC'2010* Universidade de Aveiro / IEETA 2010.